Reusing Software Design Expertise

Kevin L. Mills

INFT 960 SPRING 1994
EXPERT DATABASE SYSTEMS
School of Information Technology and Engineering
George Mason University

I. Introduction

The development of software, especially complex, real-time software or general-purpose software intended for wide applicability, consumes a substantial amount of time and money. For this reason, the software industry adopted, subsequent to a NATO conference in 1969 where Doug McIroy first introduced the concept, a goal of reusing software components. Over the two and one-half decades since, software reuse has increased in some situations, but most students of the state of software-development practice agree that McIroy's original vision has yet to be achieved and that increased reuse of software components is possible and remains a goal worth pursuing. Over the same twenty-five years, students of software engineering have come to understand that all of software development involves reuse in some form. Even where software is not reused, knowledge about a problem domain or about software design or about programming constructs is most certainly reused -- drawn from books or from the heads of experienced analysts, designers, and programmers.

The present paper addresses reuse of design knowledge as applied to development of real-time, concurrent software. The thesis is that design knowledge (from experienced designers and from textbook design methods) can be codified as knowledge in a design assistant, and that such a design assistant can be coupled to a domain analysis and modeling technique to improve the ability of inexperienced designers to produce competent, concurrent designs for real-time software. The approach advocated in this paper is unique in several facets. First, the proposed approach distills heuristics from several real-time, design methods into a set of expert rules. This approach has previously only been tried with transform analysis from structured analysis and design. Second, the proposed approach

couples the design assistant rules to a domain modeling technique. The result is a generator that can help to produce specific designs from domain models. Most other domain modeling approaches either: 1) attempt to match domain model outputs to preexisting designs or 2) to transform domain models to software components. Third, the proposed approach can be used with any domain modeling technique, or with any object-oriented analysis method, that can represent the analysis as a network of objects (possibly connected to external devices) that communicate by passing messages. As a result of this trait, the proposed approach can be applied to assist in generating concurrent designs from any object-oriented analysis method. Fourth, the proposed approach can help to identify essential information that the analysis failed to supply, can elicit the missing information, and then can facilitate the retention of that information for future use. Most other approaches based on domain analysis and modeling require that the analyst acquire all information needed for subsequent transformations before such transformations are conducted. Fifth, the proposed approach produces a representation of a concurrent design that is independent of any specific purpose. With appropriate tools, the resulting design can be represented graphically or printed, can be analyzed for performance characteristics, can be simulated to assess the function and performance of the design, or can be translated, either automatically or by hand, into an implementation. Most other approaches result in specific representations intended for specific purposes.

A strategy underlying the reuse of design knowledge, as proposed in this paper, appears in Section IV, *Automating The Reuse Of Design Knowledge.* In addition to a strategy, Section IV presents several general and specific goals that motivate the proposed approach. Following this discussion of goals, the section identifies specific research problems that must be

solved in order to achieve the goals. The section closes with a consideration of the benefits expected to accrue from achieving the goals.

To provide a better technical understanding of the proposed approach, Section V contains a small example based on a simple, real-time application. The example defines and demonstrates selected design heuristics for structuring (e.g., active-object identification, passive-object assignment, and active-object cohesion) and for defining task interfaces (e.g., identifying inter-task messages, interrupts, timers, signals, and data). Appendix B contains output from a CLIPS [GIAR93, GIAR89] implementation of the example. Since the example uses only a subset of the possible heuristics, Appendix A proposes a more complete set of design rules. Section V closes with a consideration of other issues that should be addressed for a complete treatment of the expertise needed to design concurrent software.

Prior to presenting the main points of the paper, covered in Sections IV and V, two sets of background material are included. Section II, *Software Reuse*, describes briefly the motivation behind software reuse and the kinds of reuse that can occur during software develop. A more extensive discussion discloses the range of problems that inhibit successful reuse. The section closes with a review of four recognized approaches to reuse: 1) mass-market software, 2) software design methods, 3) automated programming, and 4) domain analysis and modeling. Section III, *Related Research*, examines some specific research activities that address the reuse of software development knowledge.

A concluding section (VI) provides a summary of the ideas advanced in this paper. Software design knowledge, applicable to concurrent, real-time systems, can be codified in an expert system and then coupled to a domain modeling technique in order

to assist in the generation of concurrent designs for specific systems within a domain. That same expert system can also support generation of concurrent designs from object-oriented analyses. The example included within this paper demonstrates the feasibility of the proposed approach. The paper also outlines a series of research problems that must be solved to achieve the goals and benefits that appear possible.

II. Software Reuse

Software production comprises one of the most dynamic growth industries of the 1990's, with expansion expected into the foreseeable future; however, the productivity of software developers is not particularly great, nor is the quality of software products particularly high.

> Software production is a rapidly expanding multibillion dollar business. The products coming from this business, however, are far from satisfactory. Thus, around 1970, the term software crisis emerged. Because a crisis is something that is overcome after a limited period, the term is not used anymore; however, the problems are worse today, at least from the user's point of view. [BIBE91, p. 405]

Some inherent properties of software contribute to the low productivity and poor quality exhibited by the software industry. [BROO87] Software products are constructed from complex, custom components; such components do not scale up from a repetition of small elements in larger sizes in the manner that electronic parts can. Software products are expected to conform to the interfaces of existing human processes and systems, no matter how complex such interfaces may be. All successful software must accommodate change in order to meet new requirements and to adapt to new environments. And software remains invisible, and cannot be visualized. Since these inherent properties of software seem to impose limits on the abilities of developers to produce high-quality software at

cost-effective rates, corporations seek to maximize their investment in software development by striving to reuse software as much as possible.

Successful reuse of software components leads to increased productivity among software developers, to improved quality in the delivered products, and to more cost-effective software maintenance. [CAV89] Such improvements could prove valuable to organizations that depend on computer software. Boehm estimates that by 1995 a 20% improvement in software productivity will be worth $90 billion worldwide. [BOEH87] This estimate, made in 1987, could prove low as society's major sectors - commercial, government, and military - become increasingly reliant on software. And such reliance falls most heavily on software that already exists. Most estimates place the percentage of software development resources spent on maintenance (perfective, adaptive, and corrective) between 60-80%, and 75% of that effort goes to perfective and adaptive maintenance. [FISC92, BALZ83]

There is ample evidence to suspect that reuse can become a normal part of software development practice. For example, a study of business software systems at Raytheon Missile Systems Division found that 60% of all designs and code (in their COBOL programs) were redundant and could be reused. [LANG84] Another study of California commercial banking and insurance applications found that approximately 75% of the software functions were common to more than one program, and concluded that less than 15% of the code written for such applications is unique, novel, or specific; the remaining 85% appeared to be generic. [JONE84] While reuse targets of 60-85% appear feasible, actual results lag. For example, Matsumoto reported in 1984 that 50% of the lines of code delivered in products from the Toshiba software factory were reused; [MATSU84] and a 1989 study of NASA projects found software reuse rates of only 32%. [CURT89]

Although the reuse of software components trails what might be achieved, reuse of components alone does not reveal the entire picture about reuse in software development. Prieto-Diaz defines two levels of software reuse: 1) ideas and knowledge and 2) artifacts and components. [PRIE87a] Whenever a programmer creates software he is reusing knowledge that he already possesses, whether through training, education, experience, or a combination of these. [CURT89] On a larger scale, programming projects reuse a massive amount of knowledge, including software development process knowledge. Thus, initiatives such as that of the Software Engineering Institute to document, refine, and promote improved software development processes provide examples of reuse of ideas and knowledge to develop software. Probably the most productive reuse of knowledge to develop software obtains today from reuse of trained software development personnel. [MEYE87]

Other examples of knowledge reuse for software development abound. A huge commercial market exists for books describing data structures and algorithms, and for teaching about the nature and application of those algorithms and data structures. [STAN84] Another example of knowledge reuse is adoption of and adherence to technical standards and conventions. [RICE89] Going even further toward tangible knowledge, buying commercial software, including so-called 4GLs, can be viewed as reuse of knowledge and ideas. [BOEH87] Brooks describes a burgeoning, mass market for software programs that can be applied to specific tasks, and he proposes to:

> equip the computer-naive intellectual workers ... with personal computers and good ... writing, drawing, file, and spreadsheet programs and then [to] turn them loose. The same strategy, carried out with generalized mathematical and statistical packages and some simple programming capabilities, will also work for ... laboratory scientists. [BROO87, p. 16-17]

Introduction of commercial software products blurs the line between knowledge and artifacts. Since software artifacts and components embody ideas and knowledge, the reuse levels introduced by Prieto-Diaz perhaps have more to do with representation: knowledge and ideas being intangible until they are represented; once represented in human-readable form, they become artifacts, and when they reach a machine-executable form they can be considered software components. Between these two extremes of artifacts and components, expert systems permit knowledge to be captured, represented, and used to assist human developers to perform the tasks necessary to produce software. The key point of the preceding discussion is that software developers need to reuse more than code. (In fact, it is difficult to define reusable components apart from a context; and a context can include the requirements, a specification, a system architecture, another program or software subsystem, and a test plan and test cases. [CALD91]) The reuse of components always includes the reuse of knowledge; and knowledge is always required to reuse components.

The software development industry has achieved moderate success in the reuse of knowledge (through books, training, and college curricula) and has shown recent signs of success in the reuse of mass-market software applications for word-processing, for spreadsheets, for drawing, and for mathematical analysis. In the areas of specialized software, for custom applications and particularly for real-time processing, the ability of developers to reuse software (and even knowledge) is less evident. The reasons for this lack of success are many.

In a previous paper, the author divides reuse problems into four categories: 1) technical, 2) cognitive, 3) management, and 4) economic. [MILL92] For the reader already familiar with software reuse inhibitors, Table II-1 gives a list of the problems allocated to each of the four categories. For other

readers, each of the problems indicated in Table II-1 is discussed briefly below, beginning with the hard technical problems.

A key inhibitor to software reuse is the scant population of reusable components. Obtaining qualified candidates for reuse is difficult, and adapting submitted code to a reusable form is expensive. [CAVA89] Software is not often designed for reuse, and even when so designed, writing reusable code is difficult. [RAMA86, MEYE87] Code can be too specialized and often includes too many representational details. [STAN84] For example, Biggerstaff points out that:

Table II-1. Software Reuse Inhibitors Classified As Technical, Cognitive, Management, and Economic

Technical	Cognitive	Management	Economic
Population	Programmer Acceptance	Commitment	Intellectual Property Protection
Classification			
Location and Retrieval	Novice Programmer	Measurment	Marketing
Evaluation			
Adaptation	Force-fit		Return-On-Investment
Granularity			
Composition	Generalization		
Documentation and Representation			
Requirements Specification			

> [m]odules become less ... reusable the more specific they become because it is more ... difficult to find an exact match of detailed specifics. Modules subtlely encode ... specific information about a variety of things: operating systems, run-time library, hardware equipment, ... data packaging, interface packaging, and so forth. [BIGG87, p. 43]

And yet, separating a reusable software component from a specific context is difficult. [CALD91]

Another reason for the paucity of reusable components is a lack of producers. Most software development occurs on a project basis; yet projects will never be an appropriate place to create reusable software. Projects are hindered by a deadline focus, lack wide domain knowledge, and lack a reuse perspective. [CALD91] Production of reusable components is also inhibited by lack of accepted frameworks or system architectures into which components can be integrated. [WIRF90]

Assuming that a large population of reusable components exists, other problems elevate in significance. One such issue is classification. By what attributes should reusable components be described and classified to enable effective search and retrieval by potential users? Defining an approach that enables discrimination between very similar components is a particularly difficult classification problem. [PRIE87] Even if an acceptable classification is posited, locating and retrieving components would not be trivial. The search space could be immense. Helping a programmer retrieve a group of reuse candidates seems achievable, but allowing a programmer to find the closest match against stated requirements appears much more difficult. [RAMA86]

With a candidate set of reusable components in hand, the evaluation problem looms. There are two facets to this problem: how close to the requirements does each candidate match and how easily reusable is each candidate? In many cases, reusability relates not only to the component itself, but also to the degree of reuse experience that the programmer possesses. In one study, by Woodfield, 51 developers (25 from industry and 26 from a university) were given 21 software components and asked to determine if each component could be reused to satisfy a particular specification. [WOOD87] The study resulted in four

findings. First, programmers untrained in reuse could not evaluate the ability of a reuse candidate to satisfy implementation criteria. Second, programmers untrained in reuse are influenced by some issues that are unimportant and are not influenced by some issues that are important. Third, no groups of programmers could be identified as performing significantly better or worse in judging reusability. Finally, if a programmer judged that the work needed to reuse code was less than 70% of the effort required to build the code from scratch, then the component was chosen for reuse.

Having successfully selected a reusable component, programmers typically must overcome the adaptation problem. A programmer must understand a component in order to modify it. [CURT89] Depending on the match between the programmer's need and the reuse component, the software might require conversion to a different operating system or programming language or hardware environment. In addition, the component interfaces might not match the interfaces expected. [NOVA92] When required to adapt reusable code, the tendency among programmers it to copy and modify. [CAVA89] To avoid copying, a number of problems must be solved. For example, who owns and has responsibility for the component? How are the components maintained and synchronized with the release of products that incorporate them? [LENZ87] How can reusable code be kept available in a form that works on multiple computing platforms? [CAVA89]

Selby investigated reuse at the National Aeronautics and Space Administration (NASA), examining 25 software systems ranging in size from 3,000 to 112,000 lines of code, and found adaptation to be an important factor affecting reuse. [SELB89] He found that modules that tended to be reused without revision had: 1) fewer calls to other modules per lines of code, 2) simpler interfaces, 3) less interaction with human users, and 4) higher ratios of comments to lines of code. Such modules were

generally small; thus, did not compose a significant part of a typical software development effort.

Selby's investigation introduces the granularity problem identified by Biggerstaff. [BIGG87] Smaller, simpler components tend to be reused more because the population is large and evaluation and adaptation are easy, though finding smaller components can be hard and the payoff is usually low. Larger components tend to be reused less often because the population is low and evaluation and adaptation are hard, though finding such components is easy and the payoff can be high. Granularity of reusable components influences the composability of the components into a whole.

To be successful, reuse schemes must provide "...robust mechanisms to insure reliable and meaningful parts composition." [RICE89, p. 125] Two different approaches exist to compose systems from components. One approach relies on standards for communication and data interchange. [JONE84] In this model, reusable components, which are assumed to be fairly large, are connected together via communication channels, and data is exchanged between components in a standard format. The second approach relies on a standard architecture into which components can be linked using a range of different mechanisms. [WIRF90, JONE84]

Two remaining technical issues merit mention: the documentation and representation problem and the requirements specification problem. Documentation requirements for reusable components are at least as rigorous as for any other software, probably more rigorous. The documentation must facilitate the understanding needed to evaluate and adapt components; for large, reusable components this is critically important, but very difficult. Documentation must include a specification, a design, a design rationale, constraints on reusing the component, and test cases. [CALD91] How should this information

be represented? Another difficulty stems from user requirements statements which trigger software reuse. Users often express their requirements in a form that can disguise cues that might otherwise trigger recognition of appropriate reuse. [CURT89]

Aside from these nine technical problems, reuse is inhibited by human nature as well. [MEYE87, CURT89, SAGE90] Experienced programmers tend to view their work as creative, and they interpret reuse as routine application of old technology. Programmers also possess a certain pride of authorship and believe that they can do the job better than others. Programmers tend to distrust software developed by those they do not know. Also, the work required to understand the code of others is not normally viewed by programmers as interesting. Programmers tend to believe that they will not get credit for work that incorporates large amounts of reusable code.

Novice programmers deserve special discussion. [CURT89] The short-term memory of humans can handle about seven (plus or minus two) concepts at one time. To overcome this limitation, experienced programmers chunk complex concepts together under labels, and then the mind can process seven labels. The labels refer to information stored in hierarchical, semantic networks in a programmer’s long-term memory. Expert programmers effectively encode new information and map, compare, and analyze that information against the broad base of knowledge that they already possess. As a result, novice programmers, who might benefit most from reusable software, are not as equipped to identify, analyze, and evaluate candidates for reuse as are experienced programmers.

While programming experience provides some advantages regarding software reuse, other aspects of experience loom as impediments. For example, programmers will often try to force the application requirements to fit a structure or pattern for which they know a solution, even if the solution fails to

satisfy some of the original specifications. [CURT89] As another example, programmers might be required to abstract general concepts out of specific implementations to form a reusable component that applies across multiple domains. When a programmer possesses too much experience in a given domain, generalizing components outside of that domain can be difficult. [MEYE87, CURT89]

Aside from programmers, managers also play a role in software reuse; unfortunately, reuse often suffers from a lack of management commitment. Building a library of reusable components takes time and costs money. Managers can seldom identify the potential for a good return on the required expense. Even when managers are inclined to establish a program, and to evaluate the results as time goes by, the measurement problem interferes. [CAVA89] What measures will demonstrate increased productivity and improved quality? If measures can be defined, then how will the necessary data be collected?

On a corporate scale, management issues are perceived in economic terms. If a company delivers software that is too general and too reusable, then management might fear losing the usual follow-on business of maintenance and enhancements. [MEYE87] In addition, individual programmers or small companies that might choose to produce reusable software components have no sure means of collecting for their efforts because their code can be easily copied and distributed. [COX92] Even if intellectual property rights could be protected, how can reusable components be marketed?

Despite these many problems, the software industry shows steady progress in reusing both components and knowledge, and additional prospects can be expected as the result of several research directions. One of the most obvious areas of progress appears in the mass markets for software running under de facto

standard operating systems (and in some cases hardware components as well). Over the past decade, two huge markets for software blossomed: 1) a market for scientific and engineering workstations (pioneered by SUN) and 2) a market for office and home computing (pioneered by IBM, Intel, and Microsoft). In each of these markets, certain de facto standards developed for operating systems (UNIX in one and DOS and Windows in the other), for user interfaces (based on graphical windows), and for certain hardware capabilities (networking attachments and hard disk capacities). As a result of these markets, substantial investments have been made throughout the software industry. A number of user-interface frameworks (for example, X-windows and MS-Windows), data communications packages (e.g., TCP/IP, DCE, and DME), and database management systems have been constructed to fit within the structures provided by these de facto standards. In addition, a wide range of applications (word processors, spreadsheets, drawing programs, and mathematical and statistical analysis packages) are available in the market to support the needs of a growing population of computer users. Already, the seeds of a new market for multimedia computing can be seen.

While this mass market provides reusable software for many people, the range of applications remains limited to those that can support office workers, students, engineers, and laboratory scientists. No such mass market exists for real-time software in applications, such as home appliance control, aircraft command and control, automobile monitoring and control, factory automation, process control, medical monitoring and measurement, retail shopping automation, telecommunications processing, and so on. The state of affairs in the marketplace for real-time software finds no consensus on operating systems (or even on the need for an operating system per se), on hardware architectures, on software architectures, or even on whether high-level

languages provide a suitable basis for building real-time systems. This lack of a mass market for real-time software exists despite the fact that the amount of money spent on such software probably exceeds that spent on more conventional applications, and also despite the fact that such systems have an increasing potential to affect the health and safety of unwitting users (that is, people who rely on the software without necessarily being aware that it exists).

Lacking a mass market for reusable components, reuse in real-time software takes other forms. Over the past fifteen years, analysis and design methods have been adapted for use in real-time software (several of these methods are described in Section III). These methods generally provide: 1) a notation for specifying requirements, 2) a notation for representing designs, 3) a process to follow to analyze user requirements statements and to produce designs, and 4) heuristics for making specification and design decisions. The latter three of these elements represent knowledge about analysis and design. Manual, software-design methods make this knowledge available for reuse by writing it down (usually in a textbook form). When a designer applies one of these methods to a specific problem, he then reuses the knowledge embodied in the method. In some cases, design methods are supported by so-called computer-aided software engineering (CASE) tools that help a user to represent the analysis and design in the proper notation. Usually the CASE tool can check for consistency and completeness between the two representations. These tools provide no other assistance for the process of specifying the problem or for designing a solution.

Some research approaches attempt to further automate the design process. These approaches (specific examples are cited and explained in Section III) generally fall into three categories: 1) automatic programming, 2) end-user programming,

and 3) design assistants. Automatic programming approaches attempt to move from a formal statement of user requirements to working software without requiring additional human intervention. To accomplish such a goal, many kinds and levels of knowledge must be captured, represented, and used. In the most ambitious automatic programming systems, this knowledge can include domain-specific knowledge, programming knowledge, mathematical knowledge, hardware knowledge, and even common sense. In most automatic programming systems, the problem specification really turns out to be a statement of a solution method. [BROO87] There are exceptions when the problem domain can be characterized by a small number of parameters, where many known solutions can populate a library of alternatives, and where extensive analysis yields specific rules for selecting a solution for the given set of problem parameters.

Rather than creating a solution **for** users, end-user programming systems interact **with** users to create solutions to application problems. As with automatic programming approaches, the goal is to move from user needs to working solutions without great expense and time; however, in end-user programming approaches, human intervention is expected as the solution develops. For example, the user need not write her needs in a formal language, but might instead converse with an expert assistant to help define her problem, to tentatively select a strategy for solution, to exercise the solution, and then to correct any problems that occur. This mode of operation tends to limit these approaches to applications that are interactive, that tolerate ambiguity and error, and that are not time-critical.

While automatic and end-user programming approaches attempt to improve productivity and user satisfaction by removing the analyst, designer, and programmer from the development process, automated design assistants attempt to represent knowledge about

design strategies in some form, often in an expert system, that can assist a human designer to make the many decisions needed to transform a requirements specification into an architectural design. The power of a design assistant increases with an increasingly rich knowledge-base. Most design assistants reported to date attempt to apply rules (such as transform analysis from structured design) to transform data flow diagrams (produced by structured analysis) into structure charts that represent a sequential design. The potential exists to create concurrent designs using expert assistants that represent the richer knowledge now only embedded in manual, design methods for real-time systems.

> The most powerful contribution by expert systems will surely be to put at the services of the inexperienced programmer the experience and accumulated wisdom of the best programmers. This is no small contribution. The gap between the best software engineering practice and the average practice is very wide - perhaps wider than in any other engineering discipline. A tool that disseminates good practice would be important. [BROO87, p. 15]

No currently available design assistant encapsulates the best engineering practice for real-time software design. Section IV presents a strategy for achieving this objective in concert with domain analysis and modeling.

Domain analysis attempts to generalize all systems in an application domain, that is, to produce a domain model that transcends specific applications. [PRIE87a] Although no accepted definition for representing a domain model exists, remarkable similarity can be seen among researchers regarding the content of such a model. Jacobsen and Lindstrom describe a domain model as the set of domain objects (including their attributes and functions) and the relationships between them. [JACO91] This description mirrors that of other domain analysis and modeling researchers. [ARAN89, GOMA92, ISCO88, PRIE87a] Gomaa and Iscoe each

add to their description of a domain model the set of rules that can be used to compose, generalize, and specialize domain objects. The disagreement among researchers regarding a representational form for a domain model seems to be motivated by differences in the use that each intends for the model. Some researchers, such as Gomaa and Iscoe, intend to use the domain model as input to any of a number of transformational program generators. Others, such as Prieto-Diaz, aim only to facilitate reuse of domain concepts; here, the domain model is represented as a specific, unique language that can be used by human analysts to create specifications for individual systems within the domain. Still others, such as Arango, want to represent information that will allow a domain model to trigger specific instances of reusable components. Different than all of these approaches, Jacobson and Linstrom prefer a graph representation of the domain because they aim to build a model that facilitates reasoning about system modifications.

In summary, reuse of development knowledge and software components can potentially increase the productivity of software developers and the quality of software products. Although this potential was first recognized twenty-five years ago, a host of technical, human, and economic problems have limited software reuse to date. During the past decade, the emergence of a mass market for office, home, scientific, and engineering applications has accelerated software reuse; however, in the larger realm of real-time and custom software applications no such acceleration has occurred. A number of research approaches propose to improve this situation. Some of these approaches are described and critiqued in the next section.

III. Related Research

As practiced today, software analysis and design rely heavily on reusing design knowledge represented within textbooks, university courses, and industry training programs. A number of such analysis and design methods are surveyed in the literature. [GOMA93a, YAU86] Due to the popularity of certain of these methods, some researchers propose means for automating them. Other researchers envision eliminating design methods altogether either by automatically transforming problem specifications to implementations, or by helping users to interactively develop implementations without an analyst. [LOWR92, SIMO86] A main thrust of current research regarding software reuse applies a range of techniques to generate or adapt software from reusable models of application domains. These research efforts are reviewed below.

Software Design Methods

The earliest, effective analysis and design method combined structured analysis [DEMA78] with structured design [YOUR79]. Structured analysis (SA) provides a simple, yet effective, method for representing and comprehending a range of data processing systems using a process, or transform-oriented, view of a software problem. In SA, a system is seen as a series of steps that transform incoming data into outgoing data. Each step is represented by a circle (called a transform) that performs some processing (described in pseudo-code) on incoming data (denoted by directed arcs) to produce system outputs (also noted by directed arcs). Devices, processes, and people outside of the system are denoted by rectangles, while data repositories within the system are denoted by two, parallel lines. A complete set of transforms, directed arcs, parallel lines and rectangles composing a SA system description is called a data

flow diagram (DFD). A complete, SA specification can consist of a hierarchical set of DFDs because each transform at one level can be described by another DFD at the next lower level of detail. The result of applying structured analysis is a set of diagrams representing the data flow into a system, among a set of transforms within the system, and then out of the system.

Given a structured analysis specification for a system, a software designer must create a plan for a program that satisfies the specification and that can be coded by programmers. The most accepted method for creating such a plan, when the resulting program is to be sequential, is called structured design (SD). [YOUR79] Structured design identifies desirable properties of a sequential, program design, provides a set of heuristics for transforming a SA specification into such a design, and also gives a notation for representing the design. The software properties of concern in SD include: 1) module size (between 10 and 100 statements), 2) span of control (seven, plus or minus two), 3) fan-in (maximize within other constraints such as high module cohesion), and 4) scope of effect/scope of control (for any given decision, the scope of effect should be a subset of the scope of control of the module containing the decision).

In addition to a set of desirable goals, SD provides a mean to transform a DFD into a design. Before considering the transformation process, a short summary of the design notation is in order. A structured design is a hierarchical, module, structure chart. Each module is represented by a rectangle, with unconditional flow of control between a superior and subordinate module represented by a directed arc. Conditional flow of control is shown by augmenting the directed arc with a diamond. Parameter flows (both in and out) and control flag flows (both in and out) between modules are represented by directed arcs that parallel the control flow arcs. The tail of

parameter flow arrows possess hollow circles, while the tail of control flag flow arrows possess solid circles.

Given a DFD, the process of creating a structure chart requires four steps. First, the afferent (incoming) and efferent (outgoing) data transforms are identified. The effect of this analysis is to separate the DFD into branches of three kinds: input, processing, and output. Next, a first-level factoring is performed that creates a top-level (or main) module and a second-level module for each branch identified previously. The third step is to factor each branch; separate, factoring strategies are recommended for each type of branch. The final step, requires the designer to deal with any departures from the usual. For example, some branches may contain a mix of afferent and efferent transforms. As a final set a suggestions, structured design includes common verbs that can be used to assign module names.

Although structured analysis and design provide useful notations and approaches for designing many standard, software applications that admit sequential solutions, the wider range of software problems, including real-time systems and concurrent processing, can benefit from additional techniques. For this reason, researchers have developed real-time variants of structured analysis and design.

Real-Time Structured Analysis (RTSA) augments SA with additional semantics and notations to model events and control. [WARD85] The primary improvements made by RTSA include: control transforms and event flows. Control transforms, represented by circles enclosed in dashed lines, are used to encapsulate state transition diagrams (STDs) that order asynchronous events flowing into a system to control the processing and outputs of the system. In place of psuedo-code, each control transform in a RTSA specification is supplemented by a STD that describes the control functions of the transform. All inputs to and outputs

from RTSA control transforms are in the form of event flows, represented by dashed, directed arcs. Event flows can be further classified as triggers or enable/disable switches. Triggers represent events flowing into a control transform, or cause a data transform to be activated for one execution of the processing represented by the transform. Enable/disable events turn a data transform on or off. Once a data transform is enabled, the processing contained therein continues until the transform is disabled.

Given a RTSA specification, methods are needed for a designer to create a plan for a concurrent, software solution. Several researchers describe such methods. For example, Gomaa proposes a Design Approach for Real-Time Systems (DARTS). [GOMA84, GOMA93a] DARTS includes a set of heuristics for decomposing a DFD (augmented with control transforms and events, and thus called a control and DFD, or C/DFD) into a set of concurrent tasks. Further heuristics address the problem of inter-task message communication and synchronization between tasks and shared modules. Once tasks and shared modules are identified, DARTS relies on SD for designing the sequential processing required within each task. In his later writings about DARTS, Gomaa also describes a notation for representing the concurrent design. In DARTS, parallelograms denote tasks and modules are represented with rectangles. Each operation encapsulated within a module is represented by a small rectangle protruding from the larger module rectangle. DARTS also includes graphic notations for representing a range of inter-task messaging mechanisms, including: loosely-coupled messages into queues (or priority queues), tightly-coupled messages with and without reply, and event signals.

Nielsen and Shumate propose a similar approach for designing real-time systems (with Ada) from DFDs. [NIEL88, NIEL87] Nielsen and Shumate make two, restricting assumptions not made by Gomaa:

1) the system specification uses only DFDs (that is, uses SA) and 2) the target programming language is Ada. These assumptions affect the number and form of heuristics, as recommended by Nielsen and Shumate, needed to transform a DFD specification into a concurrent design. Overall, the heuristics proposed by Nielsen and Shumate can be considered a subset of the more comprehensive heuristics recommended by Gomaa.

The analysis and design methods discussed to this point are based on process-oriented techniques. Other methods, more recently conceived, rely on object-oriented techniques. One early attempt to blend object-oriented approaches with real-time structured analysis (RTSA) resulted in a Concurrent Object-Based Real-Time Analysis (COBRA) method. [GOM93a] COBRA augments RTSA by adding the ability to represent objects (in addition to data and control transforms) on C/DFDs. An object can be discerned on a COBRA diagram as a circle labeled with a noun (transforms are labeled with verbs). A circle outlined in a solid line denotes a data, algorithm or device object, while a dashed outline identifies a control object. For data, algorithm or device objects, incoming, directed arcs identify the operations supported by the object. Beyond these object-oriented additions, COBRA provides a technique called behavioral-scenario analysis to help the analyst create an effective COBRA specification.

Gomaa augments his COBRA technique with an adaptation of DARTS, now called the COncurrent Design Approach for Real-Time Systems (CODARTS), that shows a designer how to create a concurrent design and a module structure from a COBRA specification. [GOMA93a] In addition, CODARTS supports distributed designs by giving heuristics for decomposing a system into loosely-coupled subsystems that can be distributed. While the subsystem decomposition heuristics are new, CODARTS

builds on the same extensive set of task and module structuring criteria first proposed for DARTS.

Another concurrent analysis and design approach that builds on object-oriented modeling concepts is Entity-Life Modeling (ELM) proposed by Sanden. [SAND94, SAND89, SAND89a] ELM models a problem with two basic components: entities with life (called threads) and resources (called objects). This modeling approach results in a natural implementation of threads as Ada tasks and objects as Ada modules. Threads are identified by analyzing a problem in search of entities that have life. Objects are uncovered by looking for resources that are used by the entities. ELM avoids the need for heuristics to map from analysis to a concurrent design because the analysis itself produces a concurrent specification. In addition, Sanden believes that ELM results in fewer processes when compared with alternative approaches (such as those described above and the object-oriented approaches described below). As currently defined, ELM requires an execution environment where tasks share an address space. This limitation restricts ELM to non-distributed applications.

While the advent of Ada and growth in real-time applications encouraged research into analysis and design methods for concurrent software, research into abstract-data-type theory, and the evolution of related, programming languages, spurred the development of analysis and design approaches based on an object-orientation. The first such approach, object-oriented development (OOD) is reported in the literature by Booch. [BOOC91, BOOC86] Booch describes a design approach that structures a system into objects (rather than the operations that had been used up to that time in DFDs). From that point, each object is augmented with the operations supported, the attributes represented, and the relationships to other objects. Booch’s approach improves the conceptual model of systems by replacing

the use of loose functions with objects that encapsulate functions. Although Booch's target domains were primarily real-time systems, OOD provides little guidance for mapping an object-oriented design to concurrent tasks.

Since Booch's model focused on the design and coding phases of software development, he recognized a need to couple OOD to some analysis method. One such method is object-oriented analysis (OOA) as proposed by Coad and Yourdon. [COAD92, COAD91] OOA includes a set of analysis activities coupled with a notation for representing the results. The results of an OOA analysis are represented as a five layer model: 1) the subject layer, 2) the class-and-object layer, 3) the structure layer, 4) the attribute layer, and 5) the service layer. The first OOA activity examines the problem domain to identify objects and classes. Objects are abstractions that represent problem entities and that encapsulate attribute values and services (i.e., operations). A class is a set of objects that have uniform attributes and services (i.e., a class is an object type). Objects and classes tend to be stable, long-lived concepts in a problem domain. Each object and class identified is represented as a three-segment rectangle with the name of the object/class being placed in the top segment (the next two segments are reserved for attributes and services).

The second OOA activity structures the objects and classes into 1) a generalization and specialization hierarchy and 2) an aggregation hierarchy. After developing a structural model, Coad and Yourdon require the analyst to group related objects and classes and then to establish a subject (like a subsystem) representing each group. The fourth OOA activity defines attributes for each object and class.

The final activity of an OOA requires the analyst to define services for each object and class; this includes a specification of the service interface, a description of the

detailed behavior performed below each service, and a schematic showing what messages are exchange between objects. Detailed service behaviors are described by flowcharts or by state transition diagrams, depending on which is more appropriate for a particular service. Messages exchanged between objects are shown as directed arcs that are keyed to textual descriptions.

While OOA does provide a link between user requirements and a design technique such as OOD, OOA has only a limited ability to model concurrency and to specify requirements for real-time applications. Two more complete approaches exist: Object Modeling Technique (OMT) [RUMB91] and Object Lifecycles. [SHLA92]

Object Modeling Technique facilitates problem analysis through three complementary models: an object model, a dynamic model, and a functional model. The OMT object model provides a rich set concepts, including generalization and specialization, aggregation, multiple inheritance, general, keyed relationships between objects, and constraints. As one would expect, the graphic notation for describing these concepts becomes quite involved. OMT's object model adds a few extensions (constraints and general relationships) beyond OOA, but in the area of dynamic behavior modeling, the wealth of concepts that OMT can represent far surpasses those available with OOA.

OMT proposes the use of scenarios and event traces (similar to the behavioral-scenario analysis of COBRA) to establish object-to-object behavior flow. Within objects, OMT adopts the Statechart notation and semantics developed by Harel. [HARE90] Statecharts can represent concurrency both among and within objects. In addition, Statecharts allow an analyst to represent hierarchies of concurrency. OMT also attempts to treat the concepts of inheritance together with concurrency by providing some rules of thumb. For example, given a state transition diagram (STD) representing an object's behavior, a subclass of that object, inheriting the STD, may not add new states nor

transitions to the inherited STD. While such rules of thumb provide good cautions, their practical merits remain limited because a superclass is unlikely to embody the entire behavior of every subclass of an object. More realistically, subclasses will include new attributes and services that require additional states and transitions; thus, the analyst might be forced to rewrite the superclass STD to include the new states and transitions each time a new subclass is defined.

The final OMT model, the functional model, describes computations within a system. The functional model consists of multiple DFDs showing the flow of data through the system in terms of operations. The details of the OMT functional model are not well-integrated with either the object model or the dynamic model. A message exchange model, more along the lines of the one included with OOA, might yield better results.

As with Booch's OOD, the OMT provides little useful guidance for structuring the object model into concurrent tasks and modules. OMT suggests that a system architecture denoting subsystems be selected from among a set of typical architectures, or be developed uniquely for an application. Once an architecture is established, OMT recommends distributing the objects among the subsystems in a fashion suitable for the specific problem. Perhaps OMT presupposes that the objects identified comprise a set of reusable code that can be allocated into a set of reusable architectures. (Later sections of this review of related research describe some proposals for such an approach.)

The Object Lifecycles (OLC) method advocated by Shlaer and Mellor encompasses an approach to object-oriented analysis that results in a design and implementation that emulates the analysis. In a fashion similar to OMT, the OLC method yields three models of a system: the information model, the state model, and the process model. The information model divides a

problem into subsystems that contain objects and any associated attributes, services, inheritance, and other relationships. Three documents comprise an information model. The information structure diagram (sometimes called an entity-relationship diagram) graphically depicts the overall relationships between objects. The object and attribute descriptions and the relationship descriptions list and define each object, and their attributes, and the relationships between objects.

The OLC state model represents each object in terms of states and transitions between states. The state model representation uses a form of state transition diagram (STD), comparable to the kind used in RTSA and COBRA. In OLC, events can come from external sources, from other objects, or from system timers. OLC recommends that inheritance from the information model can be combined with the state model by restricting changes in the state-based behavior of each instance of an object. For example, an object's behavior can be defined by the superclass and subclass at creation time and remain fixed thereafter. Alternatively, OLC allows the state-based behavior of a subclass to be formed by using a subset of the superclass behavior. For special cases, OLC allows an analyst to document alternative behaviors for different subclasses in a single, superclass STD, although this violates to a great degree the concept that a superclass should possess no knowledge of specific subclasses. In addition to the internal behavior of objects, OLC enables the analyst to describe the dynamic behavior of relationships between objects.

To raise the consideration of dynamic behavior to a system level, OLC provides a model for object communication. This model represents objects and the asynchronous messages and parameters exchanged between objects. The model, used to follow the system's response to arriving external events, can form the basis of a simulation of system-wide behavior.

The process model comprises the third system view facilitated by OLC. Here, the focus is on the details within actions (that is, the transitions associated with entering particular states in a STD); these details are represented using DFDs. Coupled with the process model is an object access model that depicts the synchronous messages and parameters exchanged between state models and internal, object data.

To transform an OLC analysis model into an object-oriented design, Shlaer and Mellor advocate mapping the model onto a preexisting, low-level, run-time model. The required, run-time model must support five concepts: 1) a main program, 2) a finite-state machine, 3) a transition, 4) a timer, and 5) an active instance (i.e., task). With such underlying mechanisms, implementing the system becomes an exercise in emulating the analysis model. Each task, perhaps activated by a timer, encapsulates a finite-state machine that selects an event, identifies a transition to fire, invokes the actions associated with the transition, and then selects another event. Shlaer and Mellor identify active classes as those where a state machine is associated with each instance; other classes are passive. Within each active class (and appropriate passive classes) the passive behavior of a state machine is represented by an "assigner" class. All messages sent between active classes are assumed to be asynchronous and also of equal priority. In addition, all active classes are called from the main program which presumably acts as a task switcher, enforcing any priority requirements.

From the foregoing review of research related to software design methods several observations seem relevant. First, object-oriented techniques provide more powerful, problem-modeling facilities than earlier, process-oriented techniques. This suggests that most domain analysis and modeling methods will incorporate object-oriented techniques for

analysis and specification. Second, heuristics, originally developed for mapping between control and data flow diagrams and concurrent tasking architectures, must be adapted to work from object-oriented specifications. The only means of making this mapping to date require, as with ELM or OLC, that active objects be identified during the analysis. ELM also requires that inter-task messaging and synchronization issues be considered during the analysis, while OLC assumes that all message exchanges between active objects are asynchronous. Third, software analysis and design methods target specific systems, rather than families of systems. (Below, a review of domain analysis and modeling research identifies analysis approaches that can be applied to families of systems.)

Automating Software Design Methods

Some researchers have proposed (semi-)automated mechanisms for transforming a requirements specification into a software design. Three such approaches described in the literature involve transforming data flow diagrams (that is, a structured analysis specification) into structure charts (representing a sequential design).

Tsai and Ridge describe a Specification-Transformation Expert System (STES) that automatically translates a software requirements specification (expressed as DFDs) into a sequential design (expressed as structure charts). [TSAI88] The STES, implemented using the OPS5 expert-system shell, encapsulates the structured design method of Yourdon and Constantine within rules. The DFDs and the structure charts in STES are represented as structured facts. STES uses several textbook heuristics, including coupling, cohesion, fan-in, and fan-out, to guide the design process. Each data flow in a DFD has an associated data dictionary entry that can be used by STES to

gauge the degree of coupling between modules in a structure chart. Determining cohesion among functions is difficult for an expert system, and so a user is consulted for information required to make inferences about functional cohesion. STES attempts to maximize fan-in and tries to achieve a moderate span of control.

STES operates as a sequential set of phases. First, the DFD is factored into afferent, efferent, and transform-centered branches. This factoring results in a top-level design for the structure chart. Second, each module at the next level of the structure chart is refined using textbook guidelines for coupling, cohesion, fan-in, and fan-out. Third, the resulting, multilevel, structure chart is rendered using a CASE system from Cadre Technologies.

A different approach to transform DFDs into a structured design is embodied in a system called Computer-Aided Process Organization (CAPO). [KARI88] The goal of CAPO is to relieve a designer from using techniques such as transform and transaction analysis to create structure charts. CAPO represents DFDs as flow graphs (nodes are transforms and edges are data flows). A flow graph is converted into six matrices: 1) an incidence matrix (showing the relationship between files and transforms), 2) a precedence matrix (showing direct, data flows between transforms), 3) a reachability matrix (showing whether an indirect path exists between pairs of transforms), 4) a partial, reachability matrix (used to determine the precedence violations needed to compute a matrix of feasible groupings), 5) a matrix of feasible groupings for transforms, and 6) a matrix of timing relationships. The set of matrices are used by CAPO to compute an interdependency weight for the links joining each pair of transforms. Using these weights, the flow graph is converted into a weighted, directed graph. The weighted graph is then

decomposed into a set of non-overlapping subgraphs using a number of cluster analysis techniques.

A third approach, described by Boloix, Sorenson, and Tremblay, to automatically transform DFDs to structure charts is based on an entity-aggregate-relationship-attribute (EARA) model. [BOLO92] Here, DFDs are described formally using an EARA model. Then, transformation rules, based on set theory, are used to convert the formal, DFD description into a formal description of structure charts. The transformation is applied at the lowest level of decomposition of the DFD. The DFD is partitioned into sets that might become corresponding partitions in a structure chart. The transformation rules, defined with a grammar, carefully draw the boundaries between the system and external entities, so that the resulting structure chart captures only the automated processes of the system. As a general guideline, transforms are mapped to modules on the structure chart. Data flows are mapped to input and output parameters in the structure chart. Then, modules are added that do not have corresponding transforms on the DFD (for example, control modules, modules for access to data stores, and accept, validation and display modules that connect flows to terminators). Certain additional modules can be created with human intervention. Once the transformations are completed, a set of rough structure charts are generated. A human analyst is required to improve the structure charts.

Boloix, Sorenson, and Tremblay report that a

> ... significant amount of research is needed in this area of transformations. More research must be undertaken on the nature of the participation of the oracle in various methodologies. ... Additional research is needed on the problem of transforming one or more source environments into one or more target environments. [BOLO92, p. 437]

They further point out that additional research is needed regarding the management of changes made to the transformed version of a specification so that traceability from the

original specification can be maintained. Finally, they indicate that methods for applying metrics to evaluate transformed designs require investigation.

A fourth approach, reported in the literature by Lor and Berry, transforms requirements into a design, but without using DFDs as the source and structure charts as the target. [LOR91] This semi-automated, knowledge-based, approach was developed by Lor as the subject of a Ph.D. dissertation within the context of the System ARchitects Apprentice (SARA), a joint development of researchers at UCLA and the University of Wisconsin. [ESTR86] Before discussing Lor's method of transformation, a brief description of SARA is in order.

The goals for SARA are six: 1) to allow reasoned consideration of hardware and software tradeoffs, 2) to support building models of a system's operating environment, 3) to separate system structure from behavior, 4) to enable early detection of design flaws, 5) to facilitate composition, implementation, and testing of designs, and 6) to assist individual designers in a manner most comfortable to them. To accomplish these goals, SARA comprises tools for modeling structure and behavior. A structure language (SL) enables designers to specify a fully-nested, hierarchical structure of modules and module interconnections (using a module interconnection language). A behavioral model (based on the UCLA Graph Model of Behavior, or GMB[1]) allows designers to specify, analyze, and simulate the dynamic operation of a design. Analysis and simulation of GMB specifications is supported by a range of tools. For example, given a GMB specification, a control flow analyzer can build reachability graphs; a GMB simulator can derive stochastic queuing models from GMB specifications.

---

[1] The UCLA Graph Model of Behavior, with appropriate restrictions, is equivalent to a Petri net model.

Lor builds on the SARA environment by providing automated assistance to help a designer transform a requirements specification into a SARA structural model and GMB[2]. Lor uses DFDs and system verification diagrams (SVDs) to specify requirements. SVDs provide a stimulus-repsonse model of behavior that Lor uses to specify interactions between subsystems within a design. DFDs are then used to specify the interior of subsystems.

Lor chose a rule-based approach for his design assistant for two reasons. First, since the current set of rules for transforming requirements into SARA designs is incomplete, locking the knowledge into a procedural program is premature. Second, the sequence of rule firings provides a natural explanation facility as to why specific design choices are made. The design assistant encompasses 21 rules for building the structural model, 59 for synthesizing the control domain, and 37 for modeling the data domain. A SARA structural model is synthesized by a direct translation of the hierarchy of DFDs; at the lowest level of decomposition, the data flows map to SARA domain primitives. A SARA GMB is created from the stimulus-response model provided by the SVDs, as well as from the DFDs.

Lor reports that his research provided a better understanding, and a methodical approach, to designing systems within the SARA environment. The rules encapsulated in the design assistant are syntactically complete because every possible requirements construct is covered. The rules are not, however, complete in the sense that alternative designs cannot be considered and that the rules cannot map each requirements element to a precise design construct. A human designer must select the target requirements specification, must answer

---

[2] The paper only describes the transformation into the GMB because that was the focus of Lor's dissertation.

queries as the design progresses (to provide needed information and to indicate preferences), and must improve the resulting design once generation is complete. Given the same requirements specification technique (i.e., SVDs and DFDs), Lor believes that his approach could be adapted to other design representations by rewriting the rule consequents; however, since the most crucial step in Lor's approach is developing formal definitions, represented by SARA constructs, for every construct in his requirements language, adapting to another design representation would require that this most crucial step be repeated.

Another automated assistant that attempts to bridge between requirements and design is Fickas' Critter [FICK92], based on an earlier tool known as Glitter [FICK90], that targets composite designs, those containing a mixture of human, hardware, and software components. Critter uses an artificial intelligence paradigm of state-based search, relying on a human user to provide the domain knowledge necessary to guide the search. Critter encapsulates only domain-independent, design knowledge. Critter and a human designer interact to develop a design to solve a domain-specific problem. To date, the results with Critter are not encouraging. Critter's limited reasoning techniques prevent its use on large software engineering problems; the analysis algorithms used in Critter are too slow for an interactive design system; Critter's knowledge-base and representation omit several classes of system design concepts.

Fischer and Helm describe experience with another form of automated, design assistant. [FISC92] Their approach is to embed interactive problem-solving tools into a knowledge-based, design environment for specific, application domains. By choosing a domain-specific approach they hope to reduce the semantic gap between the problem-specification language and the software-implementation language. Their approach supports a reuse paradigm of locate (using a CatalogExplorer tool),

comprehend (using an Explainer tool), and modify (using a Modifier tool) in the very narrow domain of graphic-plotting applications written in LISP. The three, intelligent tools operate on design objects stored in a catalog of LISP programs for plotting data. The CatalogExplorer helps to identify candidate components by asking the user a series of appropriate questions. Some very detailed knowledge is needed to distinguish between concepts within the domain. The Explainer presents examples of candidate algorithms from multiple views: code, diagram, sample plot, and text. These views operate at quite a low level. The Modifier helps a user adapt a particular component to meet the user's needs, and then to update the CatalogExplorer and the Explainer to account for the new variant.

Fischer and Helm report a number of lessons from experiences with their system. First, finding appropriate reusable concepts is difficult for a user. Subjects seeking to understand a concept asked questions only about LISP and plotting (and not about the specific, application domain in which the plotting was to take place). The graphic viewers aided comprehension (but this may be because the domain involved the plotting of data). The help texts provided by the Modifier proved insufficient to enable successful program adaptation. Users appeared to have difficulty decomposing modification tasks into a set of ordered steps.

The plan for another automated, design assistant, called the Design Apprentice (DA), is reported in the literature by Waters and Tan. [WATE91] Unlike the automated assistants discussed up to this point, the DA aims to refine an already existing high-level design into a detailed, program design. The DA works within the context of a larger system, known as the Programmer's Apprentice. [RICH88]

The Programmer’s Apprentice intends to support all phases of software development from requirements analysis through software testing. [RICH88a] The apprentice and the programmer communicate through a body of shared knowledge about programming techniques. This knowledge is stored in a library of standard clichés that represent a model for the domain of computer programming. A programmer describes a specification in a formal notation, called a Plan Calculus, and the Programmer's Apprentice reasons about the needed program and maps the plan into an implementation. To date, the Programmer's Apprentice provides a seven-layer, system of knowledge representation and reasoning, known as Cake [RICH92], a Requirement's Apprentice, and a debugging assistant.

The Design Apprentice (DA), under development by Tan, will add another component to the larger, Programmer's Apprentice. The DA, starting from a high-level design described by a human designer and from a library of commonly-used fragments of specifications, designs, and algorithms, supports programming by successive elaboration. During the elaboration process, the DA can detect simple errors of inconsistency and incompleteness in a program description. The underlying environment includes both domain-specific and domain-independent knowledge. Everything the DA knows is represented using Cake's frames and the Plan Calculus.

At the front-end of the DA, a translator will convert design descriptions (input by human designers in a LISP-like form) into a plan described with the Plan Calculus. At the back-end of the DA, a coder will convert plans, representing detailed program designs, into source code (Common LISP). The heart of the DA is a designer that will interact with a human and reason, using the services of Cake, about the transformations required to convert a high-level design into a detailed design. The DA will also

support the notion of browsing the cliché library and of retrieving clichés based on features from specifications.

The DA research attempts to answer the following question: Can an automated system succeed in selecting the correct design path out of the myriad of incorrect paths? Four approaches are envisioned, two to avoid and two to solve parts of this search problem. First, the DA operates at an abstract level to avoid dealing with unnecessary details. Second, the DA uses copious knowledge to assist in making intelligent decisions. Third, use of existing algorithms from a cliché library avoids the need to discover new algorithms. Fourth, getting help from the user forces the DA to do all reasoning in a manner that can be explained to the user.

Automatic Programming

Unlike automated, design assistants, which help a human analyst complete a single, if essential, transformation in the software development process, automatic-programming systems attempt to perform, without human intervention, every transformation required to generate a working implementation from an initial specification of user requirements. Automatic programming, as applied to domain-specific applications, was first defined in the literature by Barstow.

> An automatic programming system allows a computationally naive user to describe problems using natural terms and concepts of a domain with informality, imprecision, and omission of details. An automatic programming system produces programs that run on real data to effect useful computations and that are reliable and efficient enough for routine use. [BARS85, p. 1321]

To meet Barstow's definition, automatic programming systems must possess a range of knowledge, including: 1) application domain knowledge, 2) programming knowledge, 3) mathematical knowledge, and 4) knowledge of target architectures and languages. Representing such wide-ranging knowledge might well

require automatic programming systems to support multiple methods of knowledge representation, such as rules [MITH94, ODEL93, HAYE85], frames [FIKE85], semantic networks [HSIE93, LIM92], and object-oriented models [RASM93, RETT93, MEYE88]. From Barstow's perspective, however, domain-specific knowledge provides the key to automatic programming.

> It might ... be argued that providing domain-specific knowledge could be part of an interactive specification process. That is, the automatic programming system would initially be ignorant of the domain and the user would provide the necessary domain knowledge during the process of specifying a program; after several programs have been specified, the system's knowledge of the domain would have grown substantially. This seems to be much closer to the mark: it solves the reusability problem and helps cope with the diversity of domain knowledge. [BARS85, p. 1321]

Some domain-specific approaches are reviewed immediately in the following paragraphs, beginning with the results achieved by Barstow after more than six years of effort.

Barstow describes a system, FNIX, for automatically programming software that controls devices for logging data from oil wells. [BARS91] Device-control software must log data, must control the device, must satisfy real-time constraints, and must support concurrency and distribution. Device-control programs are of moderate size.

FNIX uses a transformational paradigm: an abstract specification is transformed repeatedly through successively more concrete stages until a compilable, source program is produced. The components of a typical transformation model include a, so-called, "wide-spectrum" language (that includes constructs for describing abstract concepts from the application domain, as well as constructs for specifying more concrete, implementation details); a set of sequential transformations; and a mechanism for controlling search. FNIX, specifically, embodies 31 transformations; five of these cross levels of abstraction, while the remainder occur within a given level. FNIX avoids the issues associated with searching by relying on a

user to decide which transformations to apply and when to apply them. The user gives this information to FNIX using a script language.

In the example problem described by Barstow 84 steps in a transformation sequence lead to a seven-line program. Given a larger specification (say 500 lines), a source program of 2,000 lines might be expected after some 10,000 transformation steps, involving 500 transform types (most of which remain to be written, along with control scripts that could number several thousand lines). Barstow states that FNIX has yet to achieve his definition of automatic programming.

Another domain-specific, automatic programming system, ELF, described in the literature by Setliff, synthesizes computer-aided design (CAD) tools that automatically route wires in very large-scale, integrated (VLSI) circuits. [SETL92] ELF must: understand various physical technologies, select an appropriate, procedural decomposition, choose algorithms and data structures, manage interdependencies, and generate efficient code. ELF includes domain-specific knowledge that is represented in a variety of forms, matched to the specific problem to which the knowledge applies.

Setliff believes that domain-specific knowledge is necessary to succeed in synthesizing software. She also states that abstraction must be applied to separate the design space into smaller problems (each focussing on design of some tightly-coupled objects within the bigger problem). Knowledge, appropriate for a given level of abstraction, must be used to prune the design space.

ELF, as implemented using OPS5, comprises about 1,300 rules that transform a user-provided specification into source code using three phases. First, the design is decomposed into modules. Then, for each module, data structures and algorithms are selected. (A detailed description of the approach to

selecting data structures and algorithms is available in another paper. [SETL91]) Finally, source code is generated.

Another automatic system for synthesizing software is SINAPSE. [KANT91] SINAPSE aims: 1) to reduce the time needed for scientists and engineers to implement mathematical models, 2) to allow natural language specification of requirements for such models, 3) to reuse existing implementations, and 4) to avoid the introduction of careless errors into the implementations. To achieve these aims, SINAPSE supports a five step process. First, a domain model is developed. Second, user requirements are transformed to the necessary mathematics that underlie the particular, physical phenomenon being modeled. Then, from the math models, specific, high-level algorithms are selected, followed by detailed algorithms. Finally, code is generated. During this process, design histories are maintained in a simple tree that allows the user to review the course of decisions and to change the course at any point. After any changes are specified by the user, the process moves ahead again from that point (i.e., no dependency graphs are used to automatically alter design decisions based on user-directed changes).

Even though SINAPSE is limited to building mathematical programs for scientific applications, a number of interesting issues, having applicability to all automatic programming systems, were identified by the developers. First, a large investment is required: 1) to model (i.e., abstract, analyze, and codify) a particular domain, 2) to generate sufficient programming knowledge, and 3) to maintain the synthesis system (including the cost of moving the code to multiple platforms, and the need to generate code for multiple architectures, particularly for parallel computers). The knowledge encapsulated throughout SINAPSE is widely dispersed and, thus, finding the correct changes required to modify the system can be challenging. Second, if the synthesis system does not produce

correct results, then the end users, despite their best intentions, will examine the target code for the cause of the errors. This approach to debugging the synthesized software, reminiscent of programmers who would modify the object code output by a compiler to compensate for errors within the compiler, can be costly, unproductive, and risky. Third, the synthesis process produces a huge, data repository that can be difficult to manage.

Another automatic, programming system, reported in the literature by Smith, is the Kestrel Interactive Development System (KIDS). [SMIT91] KIDS uses a transformation approach, augmented with domain knowledge, to convert a formal, problem specification into a working program. The technology underlying KIDS is REFINE, a commercial, knowledge-based, programming environment and language that supports first-order logic, set-theoretic, data types and operations, transformation, and pattern matching. The REFINE compiler generates Common LISP code.

To apply KIDS, an analyst must move through a multi-step process. First, a domain theory (i.e., a model) is created and written in the REFINE language. The domain theory will enable the system to reason about particular specifications in the domain. Once the domain theory exists, a specification for a particular problem in the domain is developed. The user then selects a design tactic; currently, four are supported: 1) map to a library routine, 2) divide-and-conquer, 3) global search, and 4) local search. Next, the user selects (from a menu) optimizations to apply to particular expressions within the specification. KIDS then searches for appropriate, high-level, data structures (sets and sequences are supported) and converts them into machine-oriented, data types before source code is generated. The generated code is compiled into a working program.

The example use of KIDS described in Smith's paper attacks a specific problem (the Costas array problem) in the field of sonar and radar, signal processing. Developing the domain theory and the specification take an enormous amount of effort (as with most first-order logic and set-theoretic specification techniques). This effort might prove a major impediment for problems of any real size.

End-User Programming

Distinct from automatic programming, end-user programming enables a computer-naive user to interact with an intelligent agent to select, exercise, evaluate, and modify an application program. No formal specification of requirements is needed; in fact, the user need only bring the ideas in his head to a computer terminal to begin the process. Researchers at the Digital Equipment Corporation (DEC) have developed such a system, called Easyprogramming. [MARQ92]

In outline, the DEC system maps the features of a specific application to appropriate abstract methods (i.e., control structures stored in a knowledge base), elicits expertise (including variations and exceptions), translates the expertise into a form that the selected abstract control structure can use, and then modifies and extends the application to cover changes in the application requirements. To accomplish these tasks, the DEC system comprises three tools: Spark, Burn, and Firefighter. For a better understanding of the system, each of these tools is discussed in turn.

Spark, with help from a user, sifts through a hierarchy of pre-defined control structures to select an appropriate approach for the specific application at hand, and then, by consulting with the user, customizes the selected approach. Each component in the hierarchy is characterized by a set of assumptions about

the type of inputs needed and the kind of outputs produced. Where multiple control structures appear to be appropriate, Spark queries the user to reach some conclusion on which structure would be best. If Spark cannot easily explain the source of ambiguity to the user, then Spark simply makes some default assumptions and leaves the problem for Firefighter. After completing its work, Spark calls Burn to further customize the selected solution.

Burn relies on a library of knowledge acquisition tools, one is associated with each pre-defined, control structure. Each knowledge acquisition tool knows what knowledge is required for its associated control mechanism, knows how to elicit the needed knowledge, and knows how to represent that knowledge in a form needed by the control mechanism. For example, Burn might ask the user for some solutions to an example problem and for a means of distinguishing between the solutions. After Burn acquires the necessary knowledge and configures pull-down menus for the application, Firefighter is dispatched.

No program generated by Burn will work well until it has been used for a while, and is then modified to account for forgotten or unanticipated factors. Burn programs are executed under the control of Firefighter. Firefighter is an evaluator that monitors the performance of Burn programs, detects poor results, and then queries the user to diagnose and debug the application. If a detected error results from missing or incorrect knowledge, then the knowledge acquisition tool is invoked. If the control mechanism is inappropriate, then Spark is invoked to select a new mechanism.

Firefighter employs three rather sophisticated, complementary evaluation techniques to monitor the performance of Burn programs. The first two evaluation techniques rely on specific code that is included in the control mechanisms, while the third technique is built into Firefighter. The first evaluation

technique might be called: GOOD DOG, BAD DOG. Each time the application executes, the user is queried about whether the performance was adequate. If a BAD DOG response is received, then the knowledge acquisition tool is invoked. The second evaluation technique might be called: I'VE BEEN A BAD DOG. The application monitors its own performance to detect inconsistencies and inadequate results. When such problems are detected, the user is informed and the knowledge acquisition tool is invoked. This strategy is necessary because most users will not sit still during the initial development while Burn elicits knowledge about every type of case that the program might face. Instead, Burn asks for a minimum of information to start, the application then monitors its own performance, and the user is required to provide additional knowledge as needed to resolve problems and improve the performance of the application. The third evaluation strategy might be called: I THINK YOU MIGHT NEED A HORSE. Since Spark initially selects a control mechanism by making strong assumptions on weak evidence, Firefighter must compare the application output to the assumptions in order to detect incorrect control mechanisms. When an error is suspected, Spark is invoked to suggest an alternate control mechanism.

The goal of the DEC system is to supply reusable mechanisms in a usable fashion. Marques and his colleagues plan an elaborate set of steps to evaluate progress toward their goal. To assess usability they built nine applications themselves, and then presented them to users. (At the time of the report, these applications were being evaluated by the users.) If the applications appear useful, they plan to write detailed instructions for specific application tasks and then to ask users with various levels of programming skill to build some programs to solve the tasks. Then, they will ask domain experts, who perform a task well, but manually, to create a

full-scale program using the tools. (At the time of the report, one program had been built by a user; the job took eighteen days.) As a final test, they will ask an experienced programmer to develop a full-scale, hand-coded program to solve a selected application. They will then compare the development time and utility of the hand-coded program with that of a user-developed program.

To demonstrate reusability, Marques and his colleagues need to show that new control mechanisms are not needed for each new application. (This is critical because they admit that the cost of building mechanisms and their associated knowledge acquisition tools is too large if they need a special tool for each new application.) Each of the nine programs that they developed used between two and six mechanisms; thirteen mechanisms were used altogether. Seven applications used the dialog manager, six used the select mechanism, and five used the classify mechanism.

Marques and his colleagues report that "[o]ne of [their] biggest problems is getting people to 'make contact' with Spark's activity model. People buried in the details of 'real work' have difficulty understanding generic, abstract models of their tasks unless they helped to create the models." [MARQ92, p. 29] In fact, the example given in their report, an example of sifting through the hierarchy of problem/solution models, shows a bewildering array of possibilities. More discouraging is that, upon selecting an incorrect mechanism, the user can be led through a tedious, repetitious cycle of programming by example only to be sent back to the beginning to select a more appropriate mechanism. The basic approach appears to be programming by educated guess, followed by trial and error refinement.

Marques and his colleagues have developed the most sophisticated, computer-assisted software development tools

reported in the literature to date. The tools compose and refine an application from a set of reusable components. The composition method employs knowledge encoded within the tools, coupled with knowledge elicited from a domain expert. The reusable components and the elicitation, generation, and run-time tools define an architecture into which elicited knowledge can be encoded. Instead of relying on standards to define an open architecture, the developers have constructed a closed environment.

The system produced by Marques and his colleagues meet the criteria for an automatic programming system, as defined by Barstow in 1985, with one exception. The reliability of programs produced by the DEC system cannot be assessed because a given application program is never really completed. The program continues to be refined, growing smarter, and presumably more reliable, with use.

Reuse Through Domain Analysis And Modeling

In the absence of practical automatic or end-user programming, numerous software engineering researchers advocate specifying, designing, and implementing systems in a manner that enables the results of such labors to be reused in the future; and then, once the future arrives, automated mechanisms should enable these previous investments to be reused, improving both the productivity of software developers and the quality of software products. Below, several proposed approaches to software reuse are reviewed. Each of these approaches envisions domain analysis and modeling as the initial, required investment. The approaches differ, however, in the means proposed for moving from a domain model to a working software system.

One of the first approaches to software reuse through domain modeling was Draco, proposed by Neighbors. [NEIG89, NEIG84] Draco enables analyses and designs to be reused, as well as actual software components. Domain analyses result in the definition of a domain language. The domain language, along with specified mappings onto languages for other, lower-level domains, forms a model of the domain. Using the appropriate domain language, an analyst can describe the requirements for specific systems within a domain. The requirements specification is then input to a domain language parser (one must exist for each domain language) which converts the specification to an internal form, that is, a parse-tree. To create an implementation, a series of transformations is needed; each specification in a given domain language is converted to a specification in the language of a lower-level domain, until, for each component specified, an execution model is created. In effect, the lowest-level transformation for any given concept involves a mapping from a domain language construct to an execution model construct. Once all concepts from a system specification have been mapped to constructs in the execution model, software components, representing each execution model construct, can be extracted from a reusable components library and linked to form an implementation.

Draco can be viewed as an underlying execution model, coupled with some initial mappings from low-level domain concepts to the execution model. Each domain added to the model requires an analyst to define a new, domain-specific language and mappings between that language and existing domain languages, or the execution model. This leads to bottom-up construction of a richer set of domain models that can complicate evolution and maintenance. A domain analyst must know how to describe mappings between each existing, domain-specific language and new languages that are created over time. Since these complex

mappings must be manually created each time a new domain language is defined, a Draco system might become unwieldy. In the end, Draco domain analysts become designers of translators between newly-created languages and a growing array of nonstandard, lower-level languages.

A different approach to domain-oriented software reuse is proposed by Ornburn and LeBlanc. [ORNB93] They propose, in one of four forms of component composition, to instantiate implementations from preexisting, generic architectures (within a specific domain) augmented with information extracted from component descriptions. In a second form of composition, they propose to generate the generic architecture first using a generator. In a third form, they propose to build components from higher level descriptions. In a fourth form, text from a component generator would be processed to produce an implementation. They describe several experiments with their approach using the domain of protocol handlers for a telecommunications system. In one experiment, they built a generic architecture for handling multiple instances of a protocol handler and instantiated that architecture with a component description for a specific protocol. In a second experiment, they created a description of a protocol handler suited for use with a component generator. In general, they envision that components are described in two forms: 1) a path expression and 2) a code template that implements a path expression.

As described, the Ornburn and LeBlanc approach suffers from familiar inhibitors to software reuse. How will the population of generic architectures and components be created? How will programmers locate, comprehend, and modify the components and architectures? How can management be convinced to make the investment required to build generic architectures and component generators?

A more pragmatic approach to software reuse within a domain is proposed by Arango, Shoen, and Pettengill. [ARAN93] "The high cost of recovering critical knowledge motivates our formalizing it for reuse." [ARAN93, p. 234] They believe that improvements in software quality and productivity occur when designers operate in a domain-specific workspace that consolidates information from domain analyses into an automated, information-retrieval system.

A domain-specific workspace consists of two types of databases. Technology books consolidate the best organizational knowledge available about a class of problems. Product books consolidate knowledge about individual instances of solutions to specific problems. For a system composed of 32K lines of assembly code, experience shows that between ten and twenty technology books may be required. To date, Arango, et al, have realized benefits from using their approach on projects in a manual form. (They recognize that as the amount of information grows, automated support will prove essential.)

Technology books provide the key to capture, for reuse, the results of domain analyses using a well-defined process. First, a domain analyst defines a language for specifying problems within a domain. Then, formal models are created for solutions to specific problems in the domain. Third, technology books are created to demonstrate that models of known solutions explain systems within the domain. Fourth, good designs, that map solutions to specific implementation technologies, are encapsulated within the technology book for the domain. For each design, the technology book must explicitly specify issues, assumptions, constraints, and dependencies. Finally, links between reusable, software modules and designs are encoded in the domain, technology book.

The pragmatism behind technology books stems from the lack of assumptions about any particular execution model, architecture,

programming language, or elaborate transformations. Technology books simply identify artifacts and the relationships between them and then represent them in an accessible form. Arango, Shoen, and Pettengill describe plans for an automated system, called RADIO, to automate technology books. RADIO will consist of an object-oriented database (ObjectStore), a modeling language (DOLL) for representing structures that can be indexed, and a informal portion (using FrameMaker to store texts, pictures, tables, and equations). The information contained with a technology book is human-readible and is meant to be accessed, understood, and used by a human analyst. In this way, technology books aim to address some of the hard problems surrounding reuse: classification, location, comprehension, and adaptation of components. Using technology books, any software created within an organization can be archived in a form that enhances the possibility for reuse.

A more ambitious approach to the Reuse Of Software Elements (ROSE) is described by Lubars. [LUBA91] Specifically, Lubars discusses ROSE-2, a descendant of earlier work (IDeA and ROSE-1) at the Microelectronics and Computer Technology Corporation (MCC). The general aim of these MCC efforts is to reuse software requirements and designs, adapting them to solve new problems. MCC researchers believe that architectures tend to stay stable in families of systems; such stability should allow past requirements analyses and specifications to be reused, along with key, design decisions and supporting code. To achieve these aims, designs must be sufficiently abstract to cover a family of related problems, and information must exist showing how to customize designs for specific instances.

The initial MCC effort in this area was the Intelligent Design Aid (IDeA). IDeA implemented a faceted classification scheme (as proposed by Prieto-Diaz and Freeman) for organizing and searching the reuse database. IDeA could generate

executable prototypes of designs, specified as DFDs, from a library of processes and a module interconnection language. A later MCC effort, ROSE-1, combined concepts from IDeA with the software template system (STS) and a data-type reuse system. IDeA served as a front-end and STS served as a back-end, along with a library from which DFD processes could be mapped to abstract data types and implementations. ROSE-1 could generate prototype code in C, Pascal, or Ada. These compiled prototypes proved more efficient than the IDeA prototypes (which used a system called POLYLITH).

ROSE-2 builds upon a knowledge-based refinement paradigm where user-supplied requirements guide the selection and customization of a high-level design. As reported by Lubars, ROSE-2 will automate his refinement paradigm. The system will be supported by a library of high-level design schemas and a set of refinement rules. A truth maintenance system will be used to record all dependencies between requirements and design consequences. This will enable ROSE-2 to use dependency-directed backtracking to explore alternative designs; IDeA and ROSE-1 required the user to change the initial requirements specification and reapply the refinement rules to generate alternative designs. The low-level design representation for ROSE-2 will be Petri Nets, from which several views can be generated (e.g., hierarchical structure, DFDs, control flow, and state-oriented behavior). From the literature, it appears as if a top-level design in ROSE-2 might be a requirements specification.

ROSE-2 will implement a three phase process. First, a design schema will be selected that matches the user requirements. Second, an instance of the selected schema will be instantiated from the user requirements. Third, refinements and design decisions will be applied based on user input. The current state of ROSE-2 cannot achieve the intended process for several

reasons. For one, ROSE-2 has yet to address mechanisms for selecting design schemas from a reuse library based on user specifications. For another, ROSE-2 provides no support for constructing design schemas. In fact, Lubars points out the same serious limitations for IDeA, ROSE-1, and ROSE-2: creating a population of design schemas and refinement rules remains expensive. Lubars hopes that research in design recovery and domain analysis techniques will overcome these limitations.

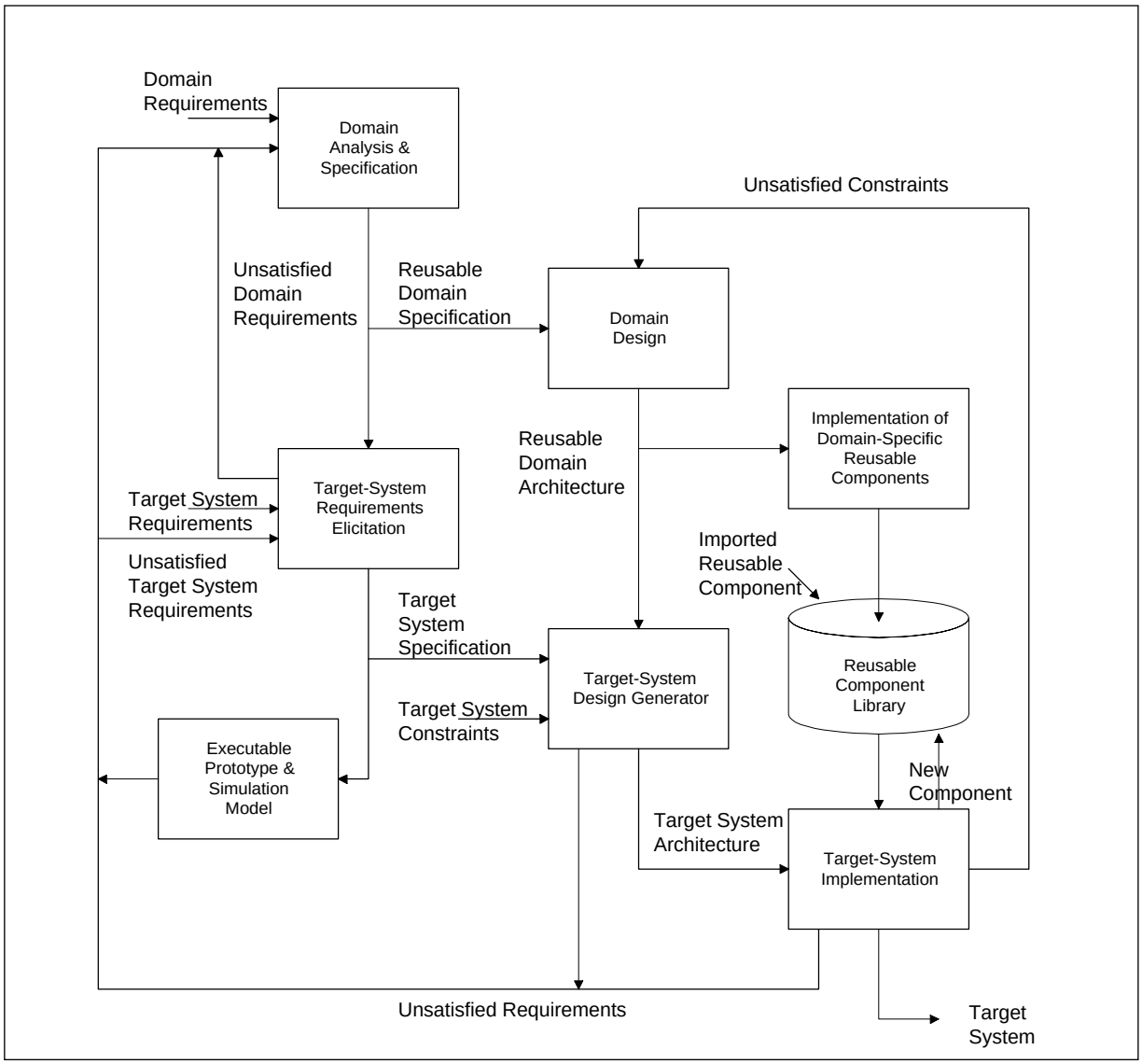

Figure III-1. Evolutionary Domain Lifecycle Model [GOMA91]

A different approach to software reuse based on domain analysis and modeling is the Evolutionary Domain Lifecycle (EDLC) model proposed by researchers at George Mason University (GMU). [GOMA93, GOMA92, GOMA91] The EDLC includes a life-cycle process (see Figure III-1), a domain modeling language, and tools for translating user requirements into an implementation (only a subset of these tools exist at present).

The first step in the EDLC process is analysis of a domain and specification of a domain model. The domain modeling language for EDLC is object-oriented, supporting several views of the domain. One view, called the aggregation hierarchy, enables an analyst to express a composition hierarchy (using the **part-of** relationship) to depict the decomposition of complex, aggregate objects (subsystems) into less complex objects, resulting at the leaves of the hierarchy in simple objects within a given domain. A second view, called the generalization/specialization hierarchy, allows a domain analyst to describe classes of objects in a domain (using the **is-a** relationship). This class hierarchy can express similarities and differences between objects. To model a specific requirement, an analyst can select the most appropriate objects from the class hierarchy or can specialize the most appropriate existing objects (updating the domain model at the same time). A third view, called object communication diagrams, enables an analyst to express the message passing relationships between objects within a domain. The object communication diagrams are structured hierarchically in concert with the aggregation hierarchy, so a picture of inter-object message exchange can be generated at any level of abstraction (even mixing such levels, if desired). The fourth view provided by the EDLC domain modeling language uses state transition diagrams. Any object within the domain that requires state-based behavior can be described using a state transition diagram. The fifth view of

EDLC ties user-oriented features in the domain model to domain objects. This feature-object dependency view indicates which objects are required for all systems in the domain, which objects are optional, which objects must be included together, and which objects are mutually exclusive. EDLC domain models can be described using a CASE tool known as Software Through Pictures (STPs); however, the semantic interpretation of the resulting data structures is formed using a separate object repository (generated with a tool written at GMU).

Once a domain model exists in the object repository, target-system specifications can be generated for a user. The user describes the features needed for the system to a knowledge-based, requirements-elicitation tool (KBRET). [SUGU93] KBRET then uses the feature-object dependencies in the domain model to extract and generate a specification that meets user needs. KBRET enforces the mandatory, inclusive, and mutually exclusive relationships, as expressed in the domain model. STP can be used to generate graphic output of the multiple-view specification extracted by KBRET.

The remaining phases of the EDLC have yet to be implemented. The EDLC vision can be seen in Figure III-1. Given a target-system specification (from KBRET), a tool should enable execution of a prototype to support functional analysis of the system; a simulation model could also be used to evaluate the performance of the system. The target-system specification might then be fed into a design generator to form a high-level architecture; presumably, EDLC envisions that design generation would be accomplished by selecting and instantiating a specific design from among a set of preexisting designs for the domain. Finally, a target-system implementation would be generated by instantiating the design from a library of reusable components.

The EDLC model provides a useful structure for developing families of systems from a domain model. The domain modeling

language takes advantage of powerful concepts from object-oriented analysis and state-based, behavior analysis. The existence of KBRET, coupled to STP, facilitates generation of target-system specifications with little effort on the user's part. As with any approach that requires a domain model, significant effort will be required to analyze and specify each new domain. If designs are to be instantiated from a set of generic designs for each domain, then additional expense will be required to populate a database of domain designs and to find a means of matching target-system specifications to existing designs. Creating the library of reusable components could also prove expensive.

IV. Automating The Reuse Of Design Knowledge

The improvements in software productivity and quality promised by domain analysis and modeling techniques have not been fully realized, especially in the domain of real-time applications. Given an appropriately analyzed domain with a fully-specified, domain model, automated mechanisms exist for generating specifications for particular systems from the domain model (which encompasses a family of systems). [SUGU93] Unfortunately, the next step in the development process, mapping a target specification to a high-level design, requires, at the present time, the intervention of skilled designers. This requirement introduces a number of impediments into the development process. First, skilled designers, especially designers of concurrent and real-time software, are a rare commodity. [BROO87] This shortage causes a bottleneck in any development process and creates a shortfall of high-level designs applicable for reuse within a domain. The shortage of skilled designers also means that few, if any, alternative designs can be considered. [BERE84] Second, designers tend to be

overused. Such overuse causes even skilled designers to make mistakes.

> Experience with large software systems shows that over half of the defects found after product release are traceable to errors in early product design. Furthermore, more than half the software life-cycle costs involve detecting and correcting design flaws. [BERE84, p. 4]

Third, designers tend to use concepts, methods, and notations that are familiar to them; thus, the ability to catalog, find, and reuse designs can be inhibited by superficial differences among designs created by various designers. [BERE84] Fourth, even skilled designers need tools to analyze designs for functional correctness and performance characteristics. Such tools are generally unavailable, but where they do exist they usually make assumptions about the design method used, or the underlying means of representing the design.

Dependence on skilled designers, lacking appropriate, automated tools, results in a design barrier that impedes effective use of domain analysis and modeling approaches for the generation and reuse of concurrent, software designs. Referring to the Evolutionary Domain Life Cycle (EDLC) model illustrated in Figure III-1, the design barrier arises once a target-system specification has been generated. In the EDLC model, a design generator is envisioned that creates a target-system architecture, presumably reusing some preexisting, domain architecture. This represents one approach to overcoming the design barrier. Unfortunately, the assumed existence of a reusable, domain architecture presents a difficulty. As pointed out be Lubars, when discussing IDeA, ROSE-1 and ROSE-2, creating a population of reusable, design schemas is expensive. [LUBA91] Lubars goes on to state that even if such a population exists, there are no agreed methods for selecting between alternative, similar, design schemas for a given requirements specification.

A second approach to overcoming the design barrier might involve addressing issues of concurrency and resource-sharing during the domain analysis and specification phase (see the first rectangle in Figure III-1). For example, a concurrent modeling method, such as Entity-Life Modeling (ELM) or Object Life Cycles (OLC), might be applied. This type of approach embodies two main drawbacks. First, domain analysis and modeling should concentrate on understanding the problem domain, not the solution domain. While identifying concurrent objects might be viewed legitimately as being within the purview of problem analysis, deciding how such objects should be packaged into tasks and modules is clearly outside the scope of problem analysis. This means that, even if ELM, OLC, or a like approach were adapted as a domain analysis and modeling method, additional, high-level, design issues would remain to be decided. Second, domain analysis and modeling should proceed independent of assumptions about the capabilities of particular target systems on which solutions can be implemented. ELM assumes that solutions will be implemented in a multi-thread environment where tasks share address space (and thus can share access to software modules). OLC assumes that concurrent objects will be represented as finite-state machines that communicate with each other via asynchronous, message passing. These, and other such, assumptions are generally inappropriate for analyzing and modeling problem domains.

A third approach to overcoming the design barrier involves mapping the domain analysis directly to a concurrent solution, where each object in the problem specification becomes a concurrent (or active) object in the design (and implementation). For example, the Actor model proposed by Hewitt and defined by Agha [AGHA90, AGHA89, AGHA87, AGHA87a, AGHA86] could be assumed, and a parallel-programming environment, such as Regis, [MAGE93] could provide a target environment.

Unfortunately, an approach such as this can result in excessive concurrency. Generally, as the number of tasks in a solution increases, the amount of overhead associated with task switching also increases. For many target environments, such task-switching overhead can become prohibitive. For other environments, such as massively parallel architectures, a multitude of tasks might prove ideal. These are decisions to be made by a designer based on performance requirements and on the capabilities of the target hardware and operating software. Automatically mapping every object in a specification to a concurrent task takes these decisions away from the designer and can lead to designs that are inappropriate in many situations.

A fourth approach to overcome the design barrier proposes assisting a designer to generate designs from a requirements specification. Several instances of this approach, limited to designs for sequential programs, were reviewed previously (in Section III). This type of semi-automated approach to design generation forms the basis for a proposal, set forth below, to overcome the design barrier inherent in domain analysis and modeling processes. The proposed approach is unique in several facets. First, the proposed approach distills heuristics from several real-time, design methods into a set of expert rules. This approach has previously only been tried with transform analysis from structured analysis and design. Second, the proposed approach couples the design assistant rules to a domain modeling technique. The result is a generator that can help to produce specific designs from domain models. Most other domain modeling approaches either: 1) attempt to match domain model outputs to preexisting designs or 2) to transform domain models to software components. Third, the proposed approach can be used with any domain modeling technique, or with any object-oriented analysis method, that can represent the analysis as a network of objects (possibly connected to external devices)

that communicate by passing messages. As a result of this trait, the proposed approach can be applied to assist in generating concurrent designs from any object-oriented analysis method. Fourth, the proposed approach can help to identify essential information that the analysis failed to supply, can elicit the missing information, and then can facilitate the retention of that information for future use. Most other approaches based on domain analysis and modeling require that the analyst acquire all information needed for subsequent transformations before such transformations are considered. Fifth, the proposed approach produces a representation of a concurrent design that is independent of any specific purpose. With appropriate tools, the resulting design can be represented graphically or printed, can be analyzed for performance characteristics, can be simulated to assess the function and performance of the design, or can be translated, either automatically or by hand, into an implementation. Most other approaches result in specific representations intended for specific purposes.

A Proposal For Reusing Design Knowledge To Generate Designs

The use of knowledge-based systems to improve human abilities to generate and evaluate designs for a range of applications (including, for example, architecture, system configuration, and building construction) is the subject of much research and great promise. [COYN90] Below, a knowledge-based strategy is proposed to overcome the design barrier that exists in domain analysis and modeling approaches to software reuse. More specifically, given:

1. 1) the EDLC model, as shown in Figure III-1,
2. 2) the domain-modeling language embedded in EDLC, and

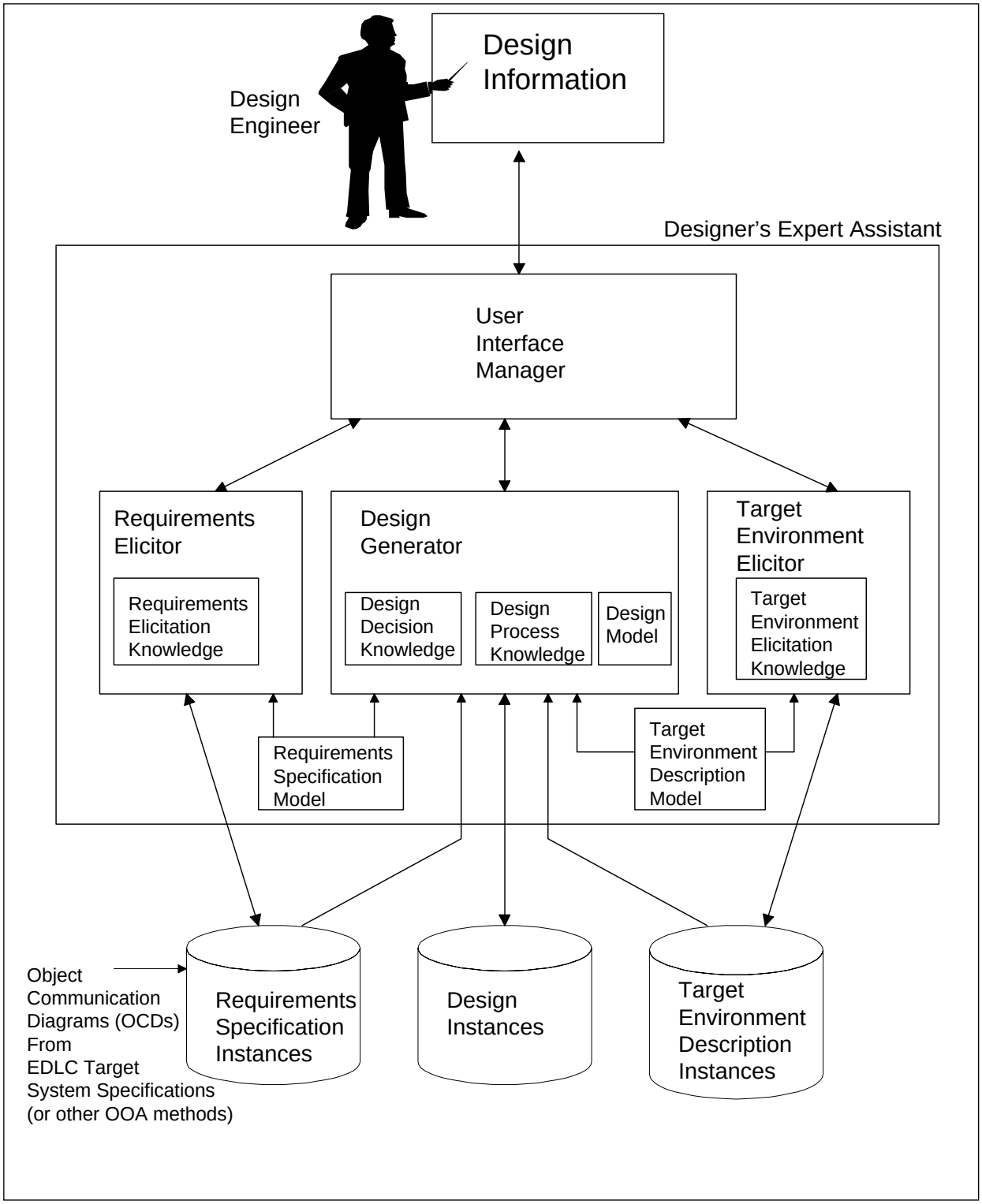

Figure IV-1. Proposed Architecture For A Designer's Expert Assistant

3. 3) KBRET (a tool for generating target systems from a domain model),

an expert system is envisioned to assist in the creation of concurrent designs for families of systems. An architecture for such an expert system is illustrated in Figure IV-1.

Requirements specifications, in the form of object communications diagrams (OCDs) that give one view of a target-system specification output from an EDLC domain model using KBRET[3], form the basis for beginning a design. The OCDs must, however, be extended with additional information to form a more complete requirements specification model. Any additional information that is not provided by the input OCDs must be obtained from an analyst or designer. The requirements elicitor fulfills this function using requirements elicitation knowledge.

Once a requirements specification exists, the designer selects a target environment for which a design is to be generated. A target environment is characterized by a description that indicates essential traits that might affect various design decisions. Should the necessary description not exist, a target-environment elicitor obtains the traits of a new, target environment.

Given a requirements specification and a target-environment description, the design generator can be invoked to formulate a structuring of tasks and modules that constitute a high-level design. The design generator must understand the models for requirements specifications, target environment descriptions, and designs. Knowledge embedded within the design generator controls a reasoning process that makes the necessary design decisions. The output of the design generator is a design,

---

[3] The initial OCDs could be generated from other object-oriented analysis methods. For example, the OOA method of Coad and Yourdon identifies message connections between objects; the Object Life Cycles approach of Shlaer and Mellor includes an object communication model; the OMT method of Rumbaugh does not provide a specific model for message communication between objects, but such a model can be derived easily from the object and functional models of OMT.

encoded in a form independent from any particular use, but a form that enables a range of possible uses.

The proposed approach, that is, assisting designers of concurrent software with an expert system, aims to achieve three general and four specific goals. Generally, the expert system should improve the productivity of inexperienced designers of concurrent software, should improve the quality and consistency of concurrent, software designs, and should overcome or avoid some software reuse problems concerning designs. Specifically, the proposed approach reuses design knowledge to assist in the critical transformation from a requirements specification to a concurrent design. The proposed approach elicits automatically any missing information needed to make design decisions. The proposed approach supports object-oriented, domain analysis and modeling methods that provide, or can be mapped to, object communication diagrams. The proposed approach ensures traceability from a requirements specification to a concurrent design.

Realizing the proposed approach requires solving some specific research problems. First, applicable design-decision heuristics must be identified. For concurrent designs, several sources exist for such heuristics. The various design methods proposed by Gomaa include a rich set of heuristics for task and module structuring. [GOMA93a] Nielsen and Shumate have also proposed some heuristics. [NIEL88, NIEL87] The Entity-Life Modeling approach developed by Sanden also contains insights regarding concurrency. [SAND94, SAND89, SAND89a] While none of these methods directly address the problem of mapping object communication diagrams to tasks and modules, many of the suggested heuristics should prove adaptable to the problem at hand.

The second research problem involves identifying and specifying the knowledge necessary to support the applicable

design heuristics. One part of this problem requires that the bare, object-communication-diagram view of the requirements specification be extended to include additional information on which design decisions can be based. The specific information that keys the decision-making of designers must be identified and specified in a form that will enable elicitation of such information. A second part of this problem requires identification of the traits of a target environment that will affect design decisions. Here, a model must be created to describe target environments. A third part of this problem requires definition of the components of a design and the relationship between these components, leading to a model for describing the generated, concurrent designs. Creating these models, for requirements specification, target-environment description, and design, will require a careful analysis and evaluation of assumptions that underlie various design methods.

A third research problem requires the identification and structuring of applicable knowledge about the design process. Once a set of design heuristics exists, the order in which decisions can be made must be considered. This will likely require that decisions be grouped into classes, according to the specific issues that they address, and then that the classes of decisions be applied in a particular order. Here, the available design methods for concurrent software should provide a starting point for considering these questions.

A fourth research problem requires the creation of elicitation strategies to obtain missing information from a designer. Strategies are required to elicit requirements information and target-environment traits. Elicitation of requirements information must be carefully considered because a user can only be asked about details that he can be reasonably expected to know. This restriction might require the requirements elicitor to reason about the basic facts provided

by a user in order to draw complex conclusions about requirements.

As each research problem is solved, a knowledge-representation method must be selected. [GONZ93, WEBS88] Representing this knowledge then becomes the fifth research problem. The requirements model, the design model, and the target-environment description model can be represented by either a semantic network [HSIE93], a frame-based notation [FIKE85], or an object-oriented model. [KAIN94, BUSC93, MEYE88] The design-decision heuristics seem to suggest a rule-based representation. [GONZ93, GIAR89 ,HAYE85] The required, design-process knowledge appears procedural. Procedural knowledge can be represented by procedural, programming languages, by mixing phase constraints into production rule systems, by employing a rule-priority scheme, and by encapsulating rules into modules whose execution is controlled by a focus stack. Each of these representations for procedural knowledge should be considered. Elicitation knowledge must also be represented. A combination of procedural knowledge and inference networks appears to be applicable to the issues faced. Procedural knowledge can control when specific questions are asked of the user, while inference networks can be used to reason about the information provided by the user.

To verify that the required knowledge is properly identified and represented, a sixth research problem must be addressed. The identified heuristics, models, processes, and elicitation strategies must be implemented to ensure that they work effectively and efficiently. Implementing the envisioned system will require an expert-system shell capable of representing procedural knowledge, rules, inference networks, and object-oriented models (or semantic networks or frames, depending on the choice for representing static models). Several candidates exist, however, the tool envisioned for

implementing the proposed, designer's expert assistant is CLIPS version 6.0. [GIAR93, GIAR89] CLIPS provides a portable environment (that runs under DOS, Microsoft Windows, Apple System 7, UNIX, and Digital's VMS), incorporates an object-oriented model together with a rule-based production system, provides for modular application of rules, and is inexpensive to procure.

Solving the research problems described above will lead to several specific benefits. First, by coupling a design generator to a domain analysis and modeling system, some serious reuse problems can be solved or avoided. One such problem is the lack of high-level designs for target specifications from a domain. A designer's expert assistant facilitates the creation of high-level designs from target specifications; thus, the lack of a design for a target specification is directly addressed by the proposed approach. The proposed approach allows one to avoid addressing criteria for classifying designs, for matching designs against target specifications, and for understanding and adapting preexisting designs to new target specifications. Even where the reuse model of classification, location, comprehension, and modification is preferred, an automated, design assistant should enable the design repository to be populated more quickly than would be possible with strictly manual approaches.

Second, generated designs, captured in automated form, can be subjected to various forms of dynamic and static analysis. For example, Sha and Goodenough show how multitasking designs can be analyzed, using rate-monotonic analysis, to determine whether timing requirements can be met under worst-case conditions. [SHA90] Smith and Williams describe a means of generating queuing models from multitasking designs to enable estimation of response time under typical system loads. [SMIT93] Pidd presents guidelines for deriving data-driven, generic simulators for

specific domains. [PIDD92] Dillon shows how to verify safety properties of multitasking programs. [DILL90]

A third benefit of the proposed approach is that the generated design will be complete and consistent with the input requirements specification. Every object and message in the requirements specification is guaranteed to be allocated to a design element, or to be identified automatically as a requirement that could not be mapped. This avoids a common source of design errors that occur when the human designer simply overlooks some of the requirements. In addition, traceability between the elements in the requirements specification and elements in the design are assured.

Automating design knowledge can help inexperienced designers to create better, more consistent, designs, faster, and with fewer mistakes. Even when designers are experienced, the increased speed with which designs can be created enables more design alternatives to be considered then would be possible without automated assistance. Considering more alternatives might result in improved flexibility, increased performance, and lowered cost for the final design. Producing designs more quickly also provides an additional benefit: formerly undetected flaws in the domain model can be revealed. This follows because decisions made during domain analysis and modeling result in object identification and structuring decisions that lead to specific, high-level, design decisions. The result of applying such design decisions to particular object communication diagrams might lead to poor designs. In such cases, the domain analyst, in consultation with a designer, can revisit the domain model to construct a more appealing structure.

The next section investigates the feasibility of the proposed approach to generate, semi-automatically, concurrent designs. In particular, a small example of a requirements specification

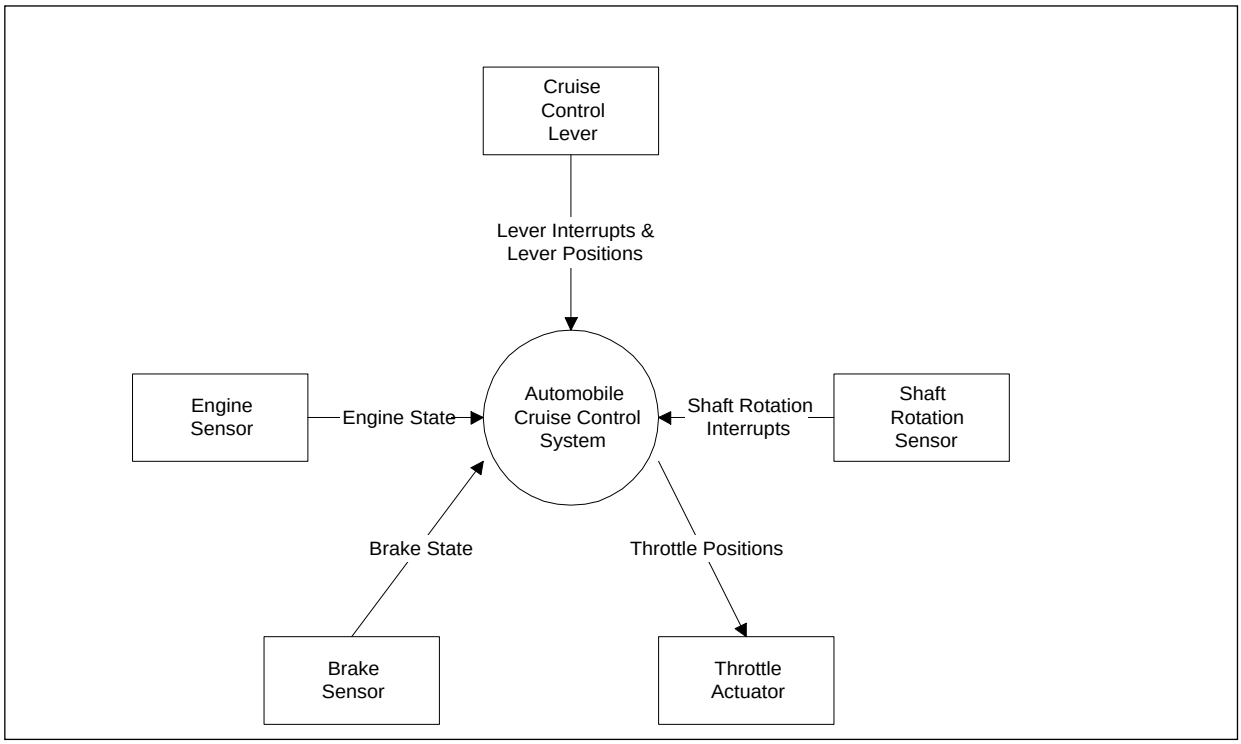

Figure V-1. Context Diagram For A Cruise Control System

for a cruise-control system is analyzed against some design heuristics that could be used to transform the requirements specification into a multitasking design. The heuristics are implemented as CLIPS rules and applied to a representation of the requirements specification to generate a design. The results suggest that the proposed approach is feasible.

V. A Case Study

This section presents a small, case study to demonstrate that the proposed approach to generating designs, as described in Section IV, appears feasible. The case study begins with a requirements specification for a cruise-control system (CCS). The context diagram, Figure V-1, for the CCS shows that the software must monitor inputs from four devices in order to

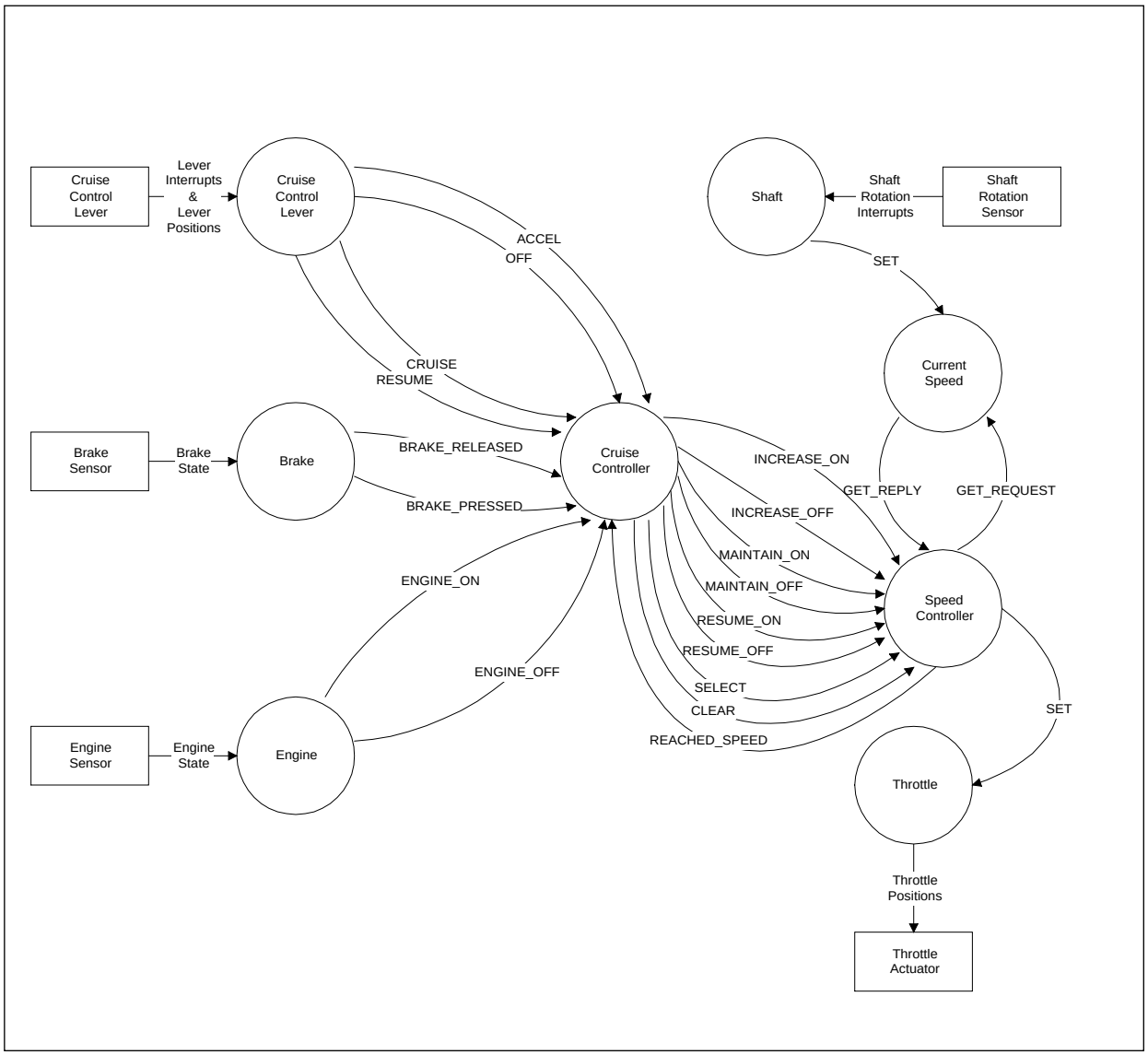

Figure V-2. Object Communication Diagram For The Cruise Control System

control the settings of a throttle actuator. The remainder of the requirements specification for the CCS is given by an object communications diagram (OCD) that decomposes the system into a set of objects that exchange messages. The OCD of most interest, shown in Figure V-2, contains all the leaf-level objects that compose the CCS, and includes the devices in the context diagram.

For this case study, a requirements model represents the OCD with some extensions. The extended, requirements model is

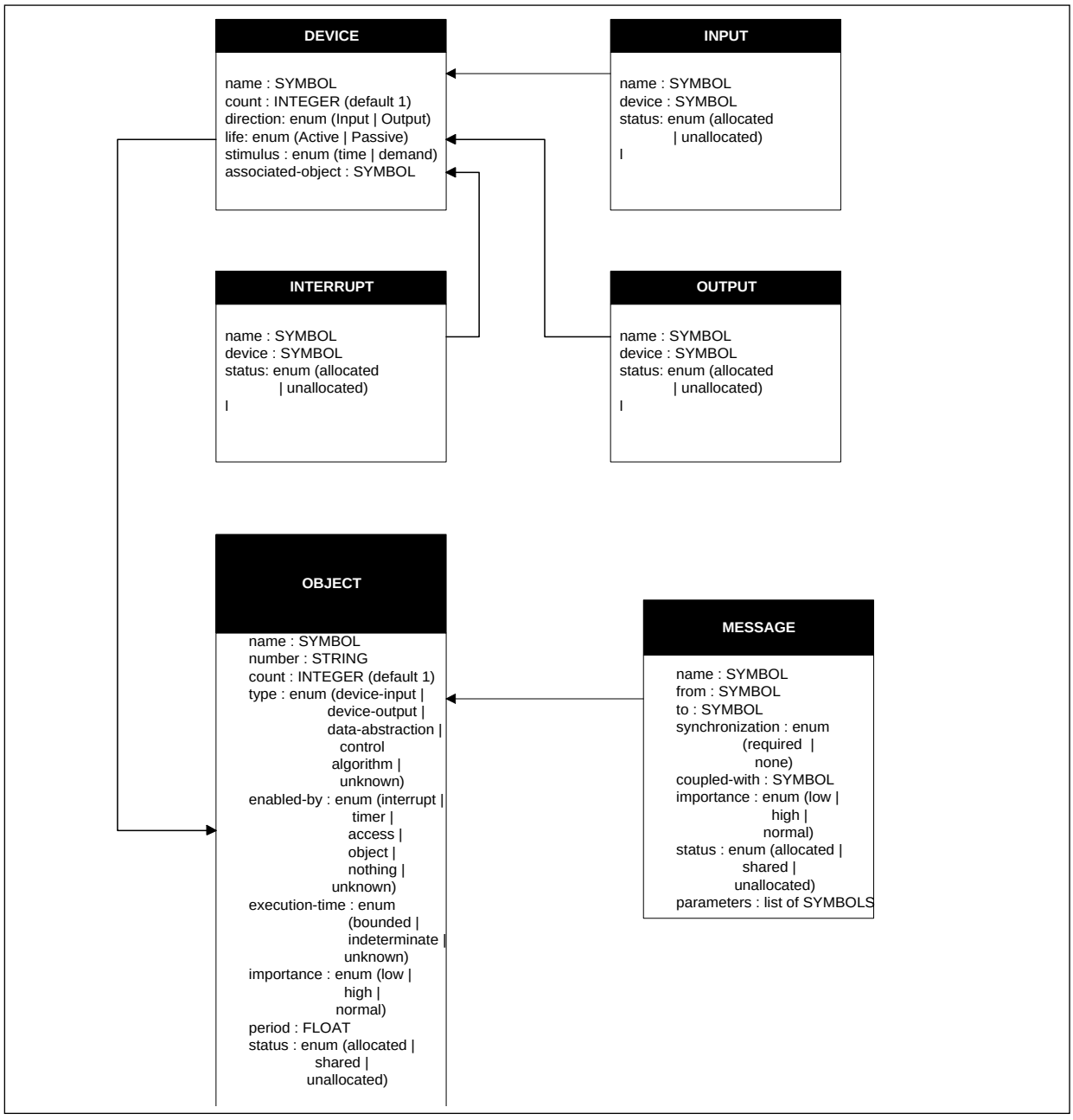

Figure V-3. Requirements Model

implemented as a set of fact templates using CLIPS. Figure V-3 illustrates the CLIPS representation of the requirements model. For devices, the name, count, direction, and associated-object

attributes come directly from the OCD. Three attributes extend the device model: 1) life indicates whether a device generates interrupts or not, 2) stimulus indicates whether a device must be accessed periodically or on demand, and 3) period gives the frequency with which a periodic device must be accessed. For objects, the name, number, and count attributes come from the OCD. Extensions enable a user to specify additional information about an object. For example, the type attribute allows the object to be classified; the enabled-by attribute provides a description of external stimulus required to trigger the object (where the object is enabled by another object, the enabler attribute names the enabling object). The user can also specify whether the execution time for the object is known to be bounded or whether the execution time cannot be determined because the object depends on varying, external conditions. The importance attribute allows an analyst to identify objects with processing of greater or lesser importance than other objects.

The message entity takes the attributes name, to, and from directly from the OCD. Some extensions enable an analyst to provide additional information. Messages can be assigned greater or lesser importance. An analyst can also identify messages that must be accepted before the sending object can continue processing. For example, in the CCS, messages sent from the Cruise-Controller to the Speed-Controller must be accepted before the Cruise-Controller continues processing because these messages are issued during transitions in a finite-state machine. The coupled-with attribute allows an analyst to identify messages that are paired with other messages. For example, in the CCS, the GET-REQUEST and GET-REPLY messages to and from the Current-Speed object are coupled because one messages replies the other.

The other portions of the requirements model used in the CCS example describe inputs, outputs, and interrupts. The

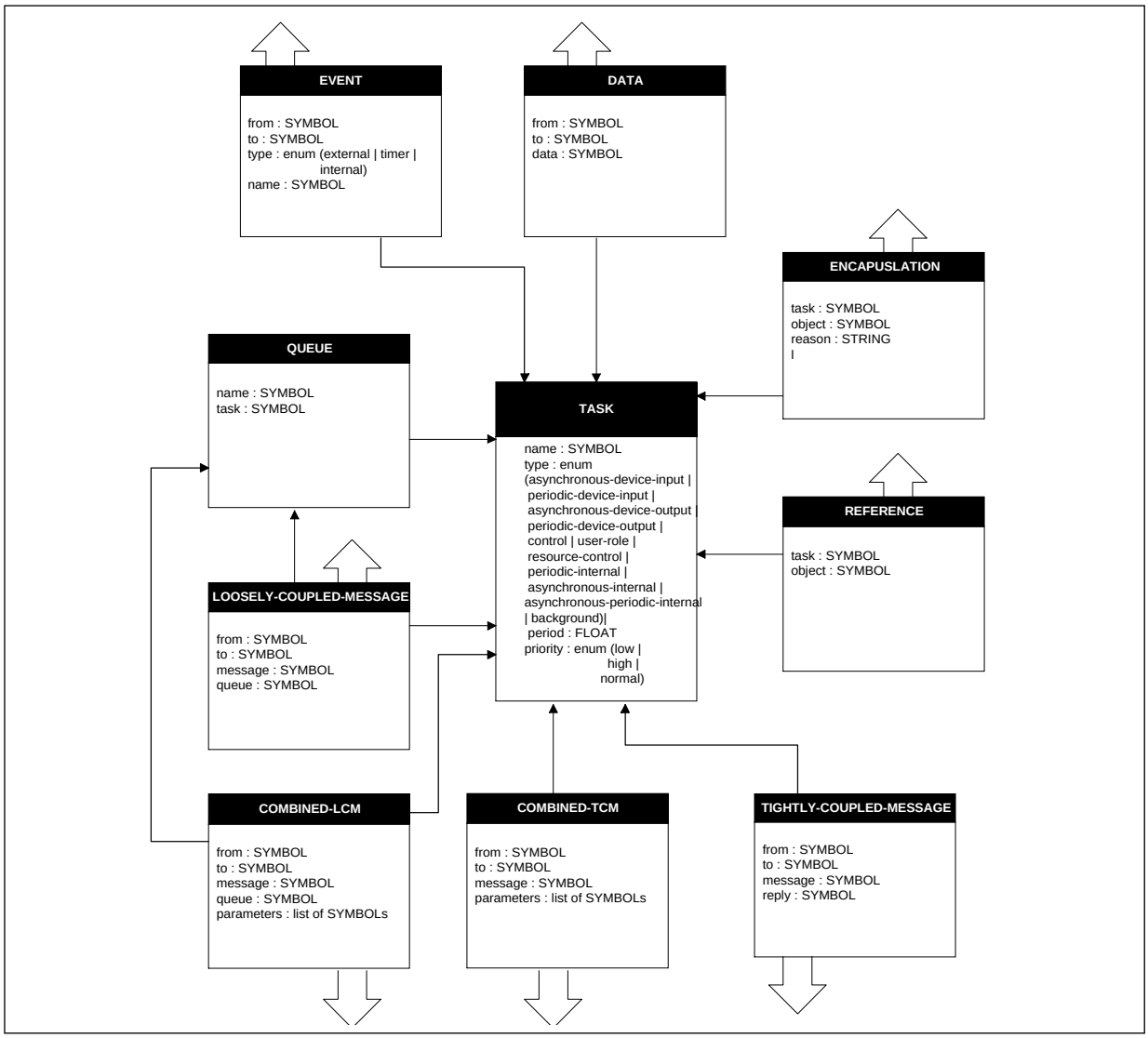

Figure V-4. Design Model

attributes for these entities come directly from the context diagram and the OCD. The directed arrows in Figure V-3 show which entities reference other entities. This simple, requirements model, while not sufficient for the system envisioned in Section IV, suffices for the CCS example.

In the example, an instance of the requirements model that corresponds to the CCS shown in Figure V-2 is used. Rather than address the elicitation of unspecified requirements for the CCS, the instantiation of the model is assumed to be complete. For

the Engine Sensor, Brake Sensor, and Throttle Actuator, 100 ms periods are used, and the corresponding objects are enabled by a timer. The Cruise Control Lever and Shaft objects are enabled by interrupts, the Current Speed and Cruise Controller objects by access, and the Speed Controller object by the Cruise Controller object. The Cruise Controller object is assigned a high importance because none of the external events affecting the CCS can be missed. Since the CCS must match the external speed of the vehicle to internal goals, the Speed Controller object has an indeterminate execution time.

In addition to completely specified requirements, other assumptions hold for the CCS example. A specific, target environment is assumed. The design will execute on a single processor, and the CCS communicates with no other subsystem. The underlying operating system is assumed to support four inter-task communication mechanisms: signals, first-in, first-out queues, shared-memory, and remote-procedure calls. Two other assumptions hold. First, the designer desires, for convenience, to compress parameterless messages between pairs of tasks into single messages that contain a type parameter. Second, the expert system should use a reasonable naming scheme to automatically assign human-readable names to the design elements that are created. These assumptions avoid the need for target environment elicitation during the CCS example.

Before discussing the heurisitics used to create a high-level, design for the CCS, the design model must be described. The design model, shown in Figure V-4, is implemented in CLIPS using facts templates. The main entity is the task (in the example considered here, module structuring is not addressed). Task attributes include a type, an optional period, and a priority. Each task includes an interface that consists potentially of queues, events, data, and messages (both loosely- and tightly-coupled). These components of the

```
R1:  if an object is a device input object and
        the associated device is Active
     then generate an asynchronous device input task

R2:  if an object is a device input object and
        the associated device is Passive and
        the associated device is Polled
     then generate a periodic device input task

R4:  if an object is a device output object and
        the associated device is Passive and
        the associated device is not accessed on demand
     then generate a periodic device output task

R5:  if an object is a control object
     then generate a control task

R9:  if an objectA is an algorithm object and
        the objectA is enabled by another objectB and
        objectA has an indeterminate execution time
     then generate an asynchronous-periodic internal task
```

Figure V-5. Active-Object-Identification Rules Used In The CCS Example

interface may be input to or output from a task. In addition, each task may enclose, or encapsulate, objects from the requirements model, and may reference other objects that are not enclosed in any task. In Figure V-4, the directed arcs show which entities in the design model refer to other entities. Each directed arc that is drawn wide, and not connected to another entity in the design model, refers to an entity in the requirements model. The design model serves for the purposes of the CCS example, but is not adequate for the more ambitious expert system outlined in Section IV.

The CCS requirements specification is transformed into a design by making a series of decisions encoded as CLIPS rules. The necessary decisions are grouped into classes with multiple rules in each class. A class of decisions is represented as a CLIPS module. Design process knowledge is represented by

applying each class of decisions in a specific order. Within each class, the order of rule execution is unconstrained. For

```
R2:  if an objectA is a data-abstraction or algorithm object
           and
        objectA is not yet allocated within a task and
        objectA receives messages from multiple objects and
        the sending objects are allocated to distinct tasks
     then denote that the objectA is shared between tasks

R3:  if an objectA is shared and
        objectA receives a message from an objectB and
        objectB is allocated to a taskJ
     then denote that taskJ accesses objectA.
```

Figure V-6. Passive-Object-Assignment Rules Used In The CCS Example

the CCS example, six classes of decisions were applied in the following order: 1) active-object identification, 2) passive-object assignment, 3) active-object cohesion, 4) task-interface-message mapping, 5) timer, interrupt, and data mapping, and 6) convenient-message combination. Only the rules

```
R1:  if taskJ and taskK (J<>K) have equal importance
        and resonating periods
     then merge the tasks into a single task
```

Figure V-7. The Active-Object-Cohesion Rule Used In The CCS Example

needed to generate a design for the CCS are coded in CLIPS for each class.[4]

Five rules, corresponding to Appendix A.1 rules R1, R2, R4, R5, and R9, are needed to identify active objects from the CCS requirements model. These rules are given in simple form in Figure V-5. Rule R1 generates a task for each asynchronous,

[4] A more complete set of rules is proposed in Appendix A. Rule numbers in the main text are keyed to those given in the appendix. The reader should understand that most of the rules in Appendix A, proposed as a result of theoretical analysis, have yet to be verified.

```
CREATE-QUEUE:  if no queue exists for taskJ and
                taskJ includes an objectA receiving a messageM
                 and messageM is not of high importance
                 and messageM does not require synchronization
               then create an input queue for taskJ

R1:  if taskJ includes objectJ and taskK includes objectK
           (J<>K)
        and objectJ receives messageM from objectK and
        messageM is not of high importance and
        messageM does not require synchronization and
        an input queueQ exists for taskK
     then allocate messageM as a loosely-coupled message from
          taskK to queueQ of taskJ

R2:  if taskJ includes objectJ and taskK includes objectK
           (J<>K)
        and objectJ receives messageM from objectK and
        messageM is not of high importance and
        messageM requires synchronization and
        messageM is not coupled with another message
     then allocate messageM as a tightly-coupled message from
          taskK to taskJ
```

Figure V-8. Rules For Mapping Messages To Task Interfaces As Used In The CCS Example

input device. In the CCS example, the Cruise Control Lever and the Shaft are such devices. Rule R2 identifies all polled, input devices (Brake and Engine) and generates a task for each.

Rule R4 generates a task for each output device that is not accessed on demand (that is, must be periodically strobed). This rule applies to the Throttle object and the Throttle Actuator device in the CCS example. Rule R5 creates a task for each control object; only the Cruise Controller qualifies in the CCS.

The final rule, R9, generates an asynchronous-periodic task (that is, a task started by some event and running periodically until disabled) for algorithm objects with an indeterminate execution time that are enabled by some other object. In the CCS, this applies only to the Speed Controller. Appendix B.1.1

```
R1:  if a task includes an object that is activated by a timer
     then allocate an input event of type timer to the task
          interface

R2:  if a task includes an object that is activated by an
        interrupt
     then allocate an input event of type interrupt to the
          task interface

R3:  if a task includes an object that receives data from a
        device
     then allocate the input data to the task interface

R4:  if a task includes an object that sends data to a device
     then allocate the output data to the task interface
```

Figure V-9. Rules Used In The CCS Example To Allocate Timers, Interrupts, And Data To Task Interfaces

shows the CLIPS decisions reached for the CCS during the active object identification step.

After a candidate set of tasks are generated for active objects, passive objects are assigned to tasks. (Although this ordering of design decisions works in the current example, assigning passive objects to tasks should be deferred until after tasks are combined, using active-object-cohesion rules, because some of the tasks to which passive objects are assigned might actually be eliminated.) For the CCS example only two rules are necessary, as shown in Figure V-6. Rule R2 simply identifies which passive objects are shared. Rule R3 ensures that the correct tasks are assigned references to shared, passive objects. In the CCS example, the data-abstraction object named Current Speed is accessed by two objects, Shaft and Speed Controller, previously allocated to separate tasks; thus, Current Speed becomes a shared object, as shown in Appendix B.1.2.

The next set of design decisions applied to the CCS attempts to merge tasks using a set of cohesion rules. Only one cohesion

```
CLCM-1:  if two distinct, parameterless, loosely-coupled
             messages are sent
           from taskJ to queueQ of taskK (J<>K)
         then create a combined, loosely-coupled message from
             taskJ to queueQ of taskK

CLCM-2:  if a combined, loosely-coupled message exists
           for taskJ, taskK, and queueQ (J<>K) and
           a loosely-coupled, parameterless messageM exists
           from taskJ to queueQ of taskK
         then merge messageM into the combined, loosely-coupled
             message

CLCM-1:  if two distinct, parameterless, tightly-couple
           messages are sent
           from taskJ to taskK (J<>K)
         then create a combined, tightly-coupled message from
             taskJ to taskK

CLCM-2:  if a combined, tightly-coupled message exists
           for taskJ and taskK (J<>K) and
           a tightly-coupled, parameterless messageM exists
           from taskJ to taskK
         then merge messageM into the combined, tightly-coupled
             message
```

Figure V-10. Rules Used In The CCS Example To Compress The Types Of Messages Exchanged Between Tasks

rule, shown in Figure V-7, applies to the CCS example. The Brake and Engine periodic tasks have a comparable importance and a resonating (in fact the same) period and, so, are merged into a single task, as shown in Appendix B.1.3.

The next step in the design process requires that the interface for each task be specified. First, messages are allocated to each task and then data, timers, and interrupts are assigned. Although these decisions are taken in separate steps for the CCS example, merging them into a single step seems possible. Figure V-8 shows the rules for task-interface-message mapping used in the CCS example. The first rule, CREATE-QUEUE,

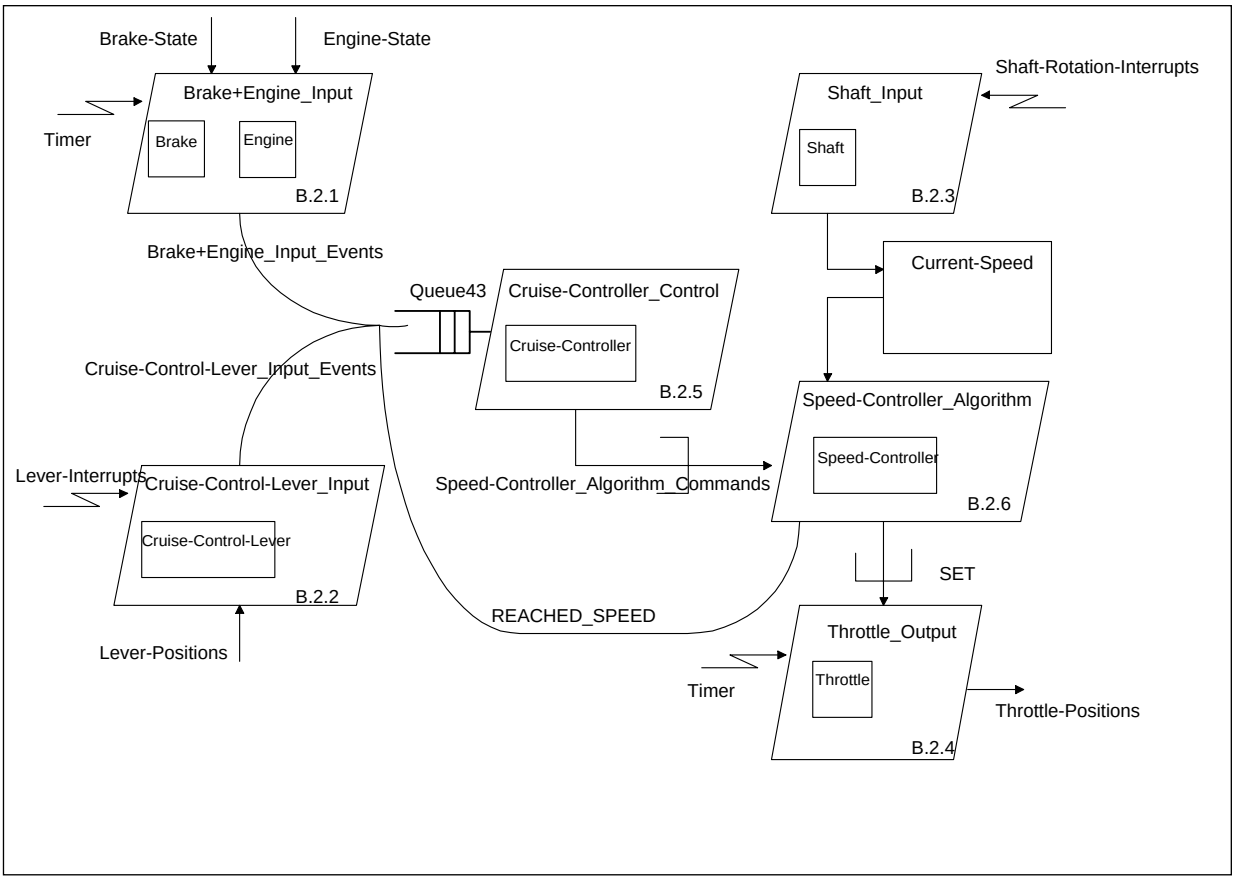

Figure V-11. High-Level, Concurrent Design For The Cruise Control System

establishes an input queue for tasks that will receive loosely-coupled messages. The second rule, R1, actually allocates appropriate messages between objects as loosely-coupled messages from the sending task into the queue of the receiving task. The final rule, R2, allocates appropriate messages between objects as tightly-coupled messages, without reply, between tasks. The results of applying these rules to the CCS problem are given in Appendix B.1.4.

The rules for mapping data, interrupts, and timers are presented in Figure V-9. These simple rules provide the housekeeping needed to map device interrupts, input and output data, and system timers to the correct task interfaces. The results of applying these rules to the CCS requirements specification are shown in Appendix B.1.5.

The final design step applied to the CCS example simply finds instances of parameterless messages flowing between pairs of tasks and then compresses those messages into single messages with a parameter to identify which of the original messages is intended.[5] This approach tends to reduce the number of messages shown on task architecture diagrams. The rules for accomplishing the compression are shown in Figure V-10. One pair of rules applies to loosely-coupled messages and the other applies to tightly-coupled messages without reply. In each case, one rule identifies the need for compression and creates a compressed message, while the second rule combines specific messages into the appropriate, compressed message. The results of executing these rules for the CCS application are shown in Appendix B.1.6.

Appendix B contains the task specifications (B.2.1 to B.2.6) produced by a CLIPS implementation of the foregoing rules applied to the cruise-control system. A task architecture diagram illustrating the high-level design generated by the expert system is shown in Figure V-11. Six tasks (represented as parallelograms) are used in the design. In the lower, right corner of each task icon a reference to the appropriate task description in Appendix B is shown. The seven objects from the requirements specification are included in the design as modules (represented as rectangles). Modules shown within a task are encapsulated by that task; shared modules (only Current-Speed in the example) are depicted outside of any task with directed arcs showing which tasks write and read to the modules. Timers and interrupts are drawn as directed, lightening flashes. Input and output data to the tasks appear in the form of directed arcs coming from or going to, respectively, the outside of each applicable task. The Cruise-Controller_Control task processes a

[5] These rules are not included in Appendix A because they are not essential to create concurrent designs.

queue (Queue43) which receives loosely-coupled messages from three tasks. The Cruise-Controller_Control task also sends tightly-coupled messages (without reply) to the Speed-Controller _Algorithm task. The Speed-Controller_Algorithm task sends tightly-coupled SET messages to the Throttle_Output task.

While this cruise-control example demonstrates that concurrent designs can be generated using heuristics represented as rules in an expert system, a number of the research problems identified in Section IV have yet to be addressed. First, the example does not deal with the elicitation of requirements information, nor with the elicitation of target-environment descriptions. Further, the case study does not address the generation of alternative designs based on target-environment descriptions. The example also does not include techniques for identifying unallocated requirements entities; nor does the example exercise the majority of the rules (see Appendix A) for identifying and merging active objects. The issue of task inversion remains to be addressed. Module structuring rules are not included, especially those needed to define module interfaces.

Other research issues, though addressed in the example, need further consideration. For example, only ad hoc knowledge of the design process is applied to the case study. Phases in the design process need clear identification so that design rules can be partitioned accordingly and so that the order of design phases can be investigated independently. Another shortcoming of the approach used in the case study involves the models. The target-environment description model is not addressed at all. While a requirements-specification model and a design model are defined for the example, several improvements are necessary. These models might better be represented using an object-oriented approach, rather than with the structured facts used in the case study. Using object-oriented models should

result in less cluttered antecedents for the design rules.[6] In addition, navigating through the models, as required in some cases, can be better accomplished using links within the model, rather than using rules.[7] No matter how each model is represented, more thought must be given to the semantics of the models. Neither the requirements-specification model nor the design model used in the example is complete. For example, the only means to explain design decisions is to review the stream of consciousness output by the CLIPS program. A better explanation facility is needed. In the requirements-specification model, the EDLC object communication diagrams are not represented faithfully. No provision exists in the requirements-specification model for representing subsystems (and no rules handle interfaces between subsystems). Finally, the design model requires a strategy for naming design elements as they are generated. In the example, a single, ad hoc naming scheme is used.

VI. Conclusions

Reuse has long been recognized as a key to improving the productivity of software developers and the quality of software products. Unfortunately, a large set of difficult problems inhibit software reuse. First among these is the low population of reusable software components, particularly architectures into which such components can be fitted. This problem bars progress in the generation of software from domain models.

The present paper proposed to reuse design knowledge, represented within an expert system, to generate concurrent designs from object-oriented, target specifications output from

[6] Using structured rules requires that each attribute needed in the rule consequent be matched in the antecedent even where those attributes play no part in the rule conditions. This tends to obscure the conditions that trigger the rule.

[7] Using rules to chain through links in a graph can become tediously complicated.

a domain model. An architecture for a designer's expert assistant was described. The architecture begins with object communication diagrams (OCDs) that form a part of the domain modeling language included in the Evolutionary Domain Life Cycle (EDLC). After eliciting missing information about the requirements and about the target-environment, the designer's expert assistant applies design-decision heuristics and design-process knowledge to transform an OCD into a concurrent design. This approach should improve the productivity of novice designers and should also increase the consistency of concurrent designs. For experienced designers, an expert assistant can facilitate the generation of alternative designs. Using the proposed approach, a repository of designs for families of systems can be populated.

A case study demonstrated the feasibility of the proposed approach. The OCD for a cruise-control system was transformed into a concurrent design using a CLIPS implementation. The example investigated: 1) a representation for the requirements and design model, 2) rules for making design decisions, and 3) methods for representing design-process knowledge. During the investigation, a number points became clear. First, object-oriented methods can provide a better representation of the requirements and design models then can the fact-based approach used in the example. Second, design decisions should be classified so that a potentially large rule set can be partitioned for easier comprehension and maintenance. Within each class of design decisions, the order of rule evaluation should be made irrelevant, if possible. Third, design-process knowledge should be implemented as an ordering among the classes of design decisions.

A number of issues were not addressed during the case study. Elicitation of requirements and target-environment information were not investigated. Interfaces between the subsystem under

design and other subsystems were not considered. Alternative designs were not permitted. Only a single, ad hoc, naming scheme was implemented to identify design elements. Although the case study verified only a subset of possible design rules, a more complete set of heuristics was given in Appendix A.

A designer's expert assistant could lead to several advantages. First, when coupled with a domain modeling and analysis method such as the EDLC, a design generator can bridge the gap between a problem analysis and a high-level design for a concurrent solution. Once a design exists, the design model can be subjected to various forms of static and dynamic analysis using automated methods to assess the function and performance of the design. Third, a generated design will be complete and consistent relative to the requirements specification, or else any omission from the requirements specification will be known explicitly. Fourth, a designer's expert assistant will codify and disseminate good design practice in a form that can help inexperienced designers create acceptable designs for concurrent software. The resulting designs should be produced faster and with fewer mistakes than would otherwise be the case. Fifth, producing designs quickly should facilitate early detection of flaws in the domain model. The earlier in the development life-cycle that errors are found, the cheaper it will be to correct them. Flaws in a domain model will be even more expensive than flaws in the analysis for a single system because the analysis within a domain model is reused more readily in many development projects. Sixth, producing alternative designs can help a designer to consider the ramifications of various decisions on the cost and performance of the resulting software. Without automated assistance, the cost of generating alternative designs can be prohibitive. Finally, automated assistance can help to generate a population of designs that can be reused on future developments.

Every problem faced by designers of concurrent software is not yet amenable to an automated solution; however, a number of design heuristics appear adaptable to encoding within an expert system. More investigation is needed to determine how best to represent requirements and designs, to adapt and verify additional design heuristics, to distill design-process knowledge into a form that can guide design decision-making, and to define factors that lead designers to select among alternatives for specific decisions.

VII. References

[AGHA90] G. Agha, "Concurrent Object Oriented Programming," *Communications of the ACM*, September 1990, pp. 125-141.

[AGHA89] G. Agha, "Foundational Issues in Concurrent Computing," *SIGPLAN NOTICES*, April 1989, pp. 60-65.

[AGHA87] G. Agha and C. Hewitt, "Concurrent Programming Using Actors," in Object Oriented Concurrent Programming, The MIT Press, Cambridge, Mass., 1987.

[AGHA87a] G. Agha and C. Hewitt, "Actors: A Conceptual Foundation for Concurrent Object Oriented Programming," in Research Directions in Object Oriented Programming, The MIT Press, Cambridge, Mass., 1987.

[AGHA86] G. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, The MIT Press, Cambridge, Mass., 1986.

[ARAN93] G. Arango, E. Shoen, and R. Pettengill, "A Process for Consolidating and Reusing Design Knowledge," in Proceedings of the 15th International Conference On Software Engineering, Baltimore, Maryland, May 17-21, 1993, pp. 231-242.

[ARAN89] G. Arango, "Domain Analysis - From Art Form To Engineering Discipline," *ACM*, 1989, pp. 247-255.

[BALZ83] R. Balzer, T.E. Cheatham, Jr., and C. Green, "Software Technology in the 1990's: Using a New Paradigm," *Computer*, November 1983, pp. 39-45.

[BARS91] D. Barstow, "Automatic Programming for Device-Control Software," in Automating Software Design, M.R. Lowry and R.D. McCartney (eds.), AAAI Press, Menlo Park, California, 1991, pp. 123-140.

[BARS85] D. Barstow, "Domain-Specific Automatic Programming," *IEEE Transactions On Software Engineering*, Vol. SE-11, No. 11, November 1985, pp. 1321-1336.

[BIBE91] W. Bibel, "Toward Predictive Programming," in Automating Software Design, M.R. Lowry and R.D. McCartney (eds.), AAAI Press, Menlo Park, California, 1991, pp. 405-424.

[BIGG87] T. Biggerstaff and C. Richter, "Reusability Framework, Assessment, and Directions," *IEEE Software*, March 1987, pp. 41-49.

[BOEH87] B. Boehm, "Improving Software Productivity," *Computer*, September 1987, pp. 43-57.

[BOLO92] G. Boloix, P.G. Sorenson, and J.P. Tremblay, "Transformations using a meta-system approach to software development," *Software Engineering Journal*, November 1992, pp. 425-437.

[BOOC91] G. Booch, Object Oriented Design With Applications, Benjamin/Cummings, Redwood City, California, 1991.

[BOOC86] G. Booch, "Object-Oriented Development," *IEEE Transactions On Software Engineering*, Vol. SE-12, No. 2, February 1986, pp. 211-221.

[BROO87] F.P. Brooks, Jr., "No Silver Bullet Essence and Accidents of Software Engineering," *Computer*, April 1987, pp. 10-19.

[CALD91] G. Caldiera and V. Basili, "Identifying and Qualifying Reusable Software Components," *COMPUTER*, February 1991, pp. 61-69.

[CAVA89] M. Cavaliere, "Reusable Code at the Hartford Insurance Group," in Software Reusability Volume II Applications and Experience, ACM Press, 1989, pp. 131-141.

[COAD92] P. Coad, "Object-Oriented Patterns," *Communications of the ACM*, Vol. 35 No. 9, September 1992, pp. 152-159.

[COAD91] P. Coad and E. Yourdon, Object-Oriented Analysis, Yourdon Press, Englewood Cliffs, NJ, 1991.

[COX92] B. Cox, The Economics of Software Reuse, a lecture given in INFT 821 at George Mason University on October 13, 1992.

[COYN90] R.D. Coyne, M.A. Rosenman, A.D. Radford, M. Balachandran, and J.S. Gero, Knowledge-Based Design Systems, Addison-Wesley, Reading, Massachusetts, 1990.

[CURT89] B. Curtis, "Cognitive Issues in Reusing Software," in Software Reusability Volume II Applications and Experience, ACM Press, 1989, pp. 131-141.

[DEMAR78] T. DeMarco, Structured Analysis and Specification, Prentice Hall, Englewood Cliffs, New Jersey, 1978.

[DILL90] L. K. Dillon, "Verifying General Safety Properties of Ada Tasking Programs," *IEEE Transactions on Software Engineering*, January 1990, pp. 51-63.

[ESTR86] G. Estrin, R.S. Fenchel, R.R. Razouk, and M.K. Vernon, "SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems," *IEEE Transactions On Software Engineering*, Vol. SE-12, No. 2, February 1986, pp. 293-311.

[FICK92] S. Fickas and R. Helm, "Knowledge Representation and Reasoning in the Design of Composite Systems," *IEEE Transactions On Software Engineering*, Vol. 18, No. 6, June 1992, pp. 470-482.

[FICK90] S. Fickas and R. Helm, A Transformational Approach to Composite System Specification, University of Oregon, CIS-TR-90-19, November 1990.

[FIKE85] R. Fikes and T. Kehler, "The Role Of Frame-Based Representation In Reasoning," *Communications of the ACM*, Vol. 28, No. 9, September 1985, pp. 904-920.

[FISC92] G. Fischer, A. Girgensohn, K. Nakakoji, and D. Redmiles, "Supporting Software Designers with Integrated Domain-Oriented Design Environments," *IEEE*

Transactions On Software Engineering, Vol. 18, No. 6, June 1992, pp. 511-522.

[GIAR93] J.C. Giarratano. et al, CLIPS Version 6.0 Reference Manuals, National Aeronautics and Space Agency, Johnson Space Center, Houston, Texas, 1993.

[GIAR89] J. Giarratano and G. Riley, Expert Systems Principles and Programming, PWS-Kent, Boston, Mass., 1989.

[GOMA93] H. Gomaa, "A Resue-Oriented Approach For Structuring And Configuring Distributed Applications," *Software Engineering Journal*, March 1993, pp. 61-71.

[GOMA93a] H. Gomaa, Software Design Methods for Concurrent and Real-Time Systems, Addison-Wesley, Reading Massachutsetts, 1993.

[GOMA92] H. Gomaa, "An Object-Oriented Domain Analysis and Modeling Method For Software Reuse," *Proceedings of the Hawaii International Conference on System Sciences*, January 1992.

[GOMA91] H. Gomaa, L. Kerschberg, C. Bosch, V. Sugumaran, and Tavakoli, " A Prototype Software Engineering Environment for Domain Modeling and Reuse," *Proceedings of the Fourth Annual Workshop on Software Resue*, November 1991.

[GOMA84] H. Gomaa, "A Software Design Method For Real-Time Systems," *Communications of the ACM*, Vol. 27 No.9, September 1984, pp. 938-949.

[GONZ93] A.J. Gonzalez and D.D. Dankel, The Engineering of Knowledge-Based Systems Theory and Practice, Prentice-Hall, Englewood Cliffs, N.J., 1993.

[HARE90] D. Harel, et al., "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transactions On Software Engineering*, Vol. 16 No. 4, April 1990, pp. 403-413.

[HAYE85] F. Hayes-Roth, "Ruled-Based Systems," *Communications of the ACM*, Vol. 28, No. 9, September 1985, pp. 921-932.

[HSIE93] D. Hsieh, "A logic to unify semantic-network knowledge systems with object-oriented database models," *Journal Of Object-Oriented Programming*, Vol. 6 No. 2, May 1993, pp. 55-67.

[ISCO88] N. Iscoe, "Domain-Specific Reuse: An Object-Oriented and Knowledge-Based Approach," an updated version of an article in the *Proceedings of the Workshop on Software Reuse* held in October 1987, pp. 299-308.

[JACO91] I. Jacobson and F. Lindstrom, "Re-engineering of Old Systems to an Object Oriented Architecture," *OOPSLA '91 onference Proceedings*, October 1991, pp. 340-350.

[JONE84] T. Jones, "Reusability in Programming: A Survey of the State of the Art," *IEEE Transactions on Software Engineering*, September 1984, pp. 488-493.

[KAIN94] H. Kaindl, "Object-oriented approaches in software engineering and artificial intelligence," *Journal Of Object-Oriented Programming*, Vol. 5 No. 8, January 1994, pp. 38-44.

[KANT91] E.Kant, F. Daube, W. MacGregor, and J. Wald, "Scientific Programming by Automated Synthesis," in Automating Software Design, M.R. Lowry and R.D. McCartney (eds.), AAAI Press, Menlo Park, California, 1991, pp. 169-206..

[KARI88] J. Karimi and B.R. Konsynski, "An Automated Software Design Assistant," *IEEE Transactions On Software Engineering*, Vol. 14, No. 2, February 1988, pp. 194-210.

[LANG84] R. Langergan and C. Grasso, "Software Engineering with Reusable Designs and Code," *IEEE Transactions on Software Engineering*, September 1984, pp. 498-501.

[LENZ87] M. Lenz, et al., "Software Reuse Through Building Blocks," *IEEE Software*, July 1987, pp. 34-42.

[LIM92] E-P. Lim and V. Cherkassky, "Semantic Networks and Associative Databases," *IEEE Expert*, August 1992, pp. 31-40.

[LOR91] K.E. Lor and D.M. Berry, "Automatic Synthesis of SARA Design Models From Systems Requirements," *IEEE Transactions On Software Engineering*, Vol. 17 No. 12, December 1991, pp. 1229-1240.

[LOWR92] M. Lowry, "Software Engineering in the Twenty-First Century," *AI Magazine*, Vol. 14 No. 3, Fall 1992, pp. 71-78.

[LUBA91] M.D. Lubars, "The ROSE-2 Strategies for Supporting High-Level Software Design Reuse," in Automating Software Design, M.R. Lowry and R.D. McCartney (eds.), AAAI Press, Menlo Park, California, 1991, pp. 93-118.

[MAGE93] J. Magee, N. Dulay, and J. Krammer, "Process Parallel Programming: A Constructive Development Environment," an unpublished manuscript, 1993, 19 pages.

[MARQ92] D. Marques, G. Dallemagne, G. Klinker, J. McDermott, and D. Tung, "Easy Programming Empowering People to Build Their Own Applications," *IEEE Expert*, June 1992, pp. 16-29.

[MATSU84] Y. Matsumoto, "Some Experience in Promoting Reusable Software Presentation in Higher Abstraction Levels," *IEEE Transactions on Software Engineering*, September 1984, pp. 502-512.

[MILL92] K. Mills, Requirements Engineering for Software Reuse, a paper produced for George Mason University doctoral seminar INFT 851, November 1992.

[MITH94] R. Mithani, "Modeling databases with objects and rules," *Object Magazine*, Vol. 3 No. 5, January 1994, pp. 58-60.

[MEYE88] B. Meyer, Object-Oriented Software Construction, Prentice-Hall, Hemel Hempstead, United Kingdom, 1988.

[MEYE87] B. Meyer, "Reusability: The Case for Object-Oriented Design," *IEEE Software*, March 1987, pp. 50-64.

[NEIG89] J. Neighbors, "DRACO: A Method for Engineering Reusable Software Systems," in Software Reliability Volume I Concepts and Models, ACM Press, 1989, pp. 295-319.

[NEIG84] J. Neighbors, "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions On Software Engineering*, Vol 10. No. 5, September 1984, pp. 564-574.

[NIEL88] K. Nielsen and K. Shumate, Designing Large Real-Time Systems with Ada, McGraw-Hill, New York, New York, 1988.

[NIEL87] K. Nielsen and K. Shumate, "Designing Large Real-Time Systems with Ada," *Communications of the ACM*, Vol. 30 No. 8, August 1987, pp. 695-715.

[NOVA92] G. Novak, et al., "Negotiating Interfaces for Software Reuse," *IEEE Transactions on Software Engineering*, July 1992, pp. 646-652.

[ODEL93] J.J Odell, "Specifying requirements using rules," *Journal Of Object-Oriented Programming*, Vol. 6 No. 2, May 1993, pp. 20-24.

[ORNB93] S.B. Ornburn and R.J. LeBlanc. Jr., "Building, Modifying, and Using Component Generators," in Proceedings of the 15th International Conference On Software Engineering, Baltimore, Maryland, May 17-21, 1993, pp. 391-402.

[PIDD92] M. Pidd, "Guidelines for the design of data driven generic simulators for specific domains," *Simulation*, October 1992, pp. 237-243.

[PRIE87] R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability," *IEEE Software*, January 1987, pp. 6-16.

[PRIE87a] R. Prieto-Diaz, "Domain Analysis For Reusability," *IEEE*, 1987, pp. 23-29.

[RAMA86] C. Ramamoorthy, et al., "Programming in the Large," *IEEE Transactions on Software Engineering*, July 1986, pp. 769-783.

[RASM93] D.W. Rasmus, "Taming the AI madness with object methods," *Object Magazine*, Vol. 3 No. 4, November-December, 1993, pp. 58-60.

[RETT93] M. Rettig, G. Simons, and J Thomson, "Extended Objects," *Communications of the ACM*, Vol, 36 No. 8, August 1993, pp. 19-24.

[RICE89] J. Rice and H. Schwetman, "Interface Issues in a Software Parts Technology," in Software Reusability Volume I Concepts and Models, ACM Press, 1989, pp. 125-139.

[RICH92] C. Rich and Y. Fedlman, "Seven Layers of Knowledge Representation and Reasoning in Support of Software Development," *IEEE Transactions On Software Engineering*, Vol. 18 No. 6, June 1992, pp. 451-469.

[RICH88] C. Rich and R.C. Waters, "The Programmer's Apprentice: A Research Overview," *Computer*, Vol. 21, No. 11, November 1988, pp. 10-25.

[RICH88a] C. Rich and R.C. Waters, "Automatic Programming: Myths and Prospects," *Computer*, August 1988, pp. 40-51.

[RUMB91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Object-Oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, New Jersey,1991.

[SAGE90] A. Sage and J. Palmer, Software Systems Engineering, John Wiley and Sons, 1990.

[SAND94] B. Sanden, Software Systems Construction With Examples In Ada, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[SAND89] B. Sanden, "Entity-Life Modeling and Structured Analysis in Real-Time Software Design - A Comparison," *Communications of the ACM*, Vol. 32, No. 12, December 1989, pp. 1458-1466.

[SAND89a] B. Sanden, "An Entity-Life Modeling Approach To The Design Of Concurrent Software," *Communications of the ACM*, Vol. 32, No. 3, March 1989, pp. 330-343.

[SELB89] R. Selby, "Quantitative Studies of Software Reuse," in Software Reusability Volume II Applications and Experience, ACM Press, 1989, pp. 213-233.

[SETL92] D.E. Setliff and R.A. Rutenbar, "Knowledge Representation and Reasoning in a Software Synthesis Architecture," *IEEE Transactions On Software Engineering*, Vol. 18, No. 6, June 1992, pp. 523-533.

[SETL91] D. Setliff, "On the Automatic Selection of Data Structure and Algorithms," in Automating Software Design, M.R. Lowry and R.D. McCartney (eds.), AAAI Press, Menlo Park, California, 1991, pp. 207-226.

[SHA90] L. Sha and J.B. Goodenough, "Real-Time Scheduling Theory and Ada," *Computer*, April 1990, pp. 53-62.

[SHLA92] S. Shlaer and S. J. Mellor, Object Lifecycles Modeling the World in States, Yourdon Press, Englewood Cliffs, NJ, 1992.

[SIMO86] H.A. Simon, "Whether Software Engineering Needs to Be Artificially Intelligent," *IEEE Transactions On Software Engineering*, Vol. SE-12 No. 7, July 1986, pp. 726-732.

[SMIT93] C.U. Smith and L.G. Williams, "Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives," *IEEE Transactions On Software Engineering*, Vol. 19 No. 7, July 1993, pp. 720-741.

[SMIT91] D. R. Smith, "KIDS - A Knowledge-Based Software Development System," in Automating Software Design, M.R. Lowry and R.D. McCartney (eds.), AAAI Press, Menlo Park, California, 1991, pp. 483-514.

[STAN84] T. Standish, "An Essay on Software Reuse," *IEEE Transactions on Software Engineering*, September 1984, pp. 494-497.

[SUGU93] V. Sugumaran, A Knowledge-Based Approach For Generating Target System Specifications From A Domain Model, Ph.D. dissertation, George Mason University, Fairfax, Virginia, 1993.

[TSAI88] J.P. Tsai and J.C. Ridge, "Intelligent Support for Specifications Transformation," *IEEE Software*, November 1988, pp. 28-35.

[WARD85] P. Ward and S. Mellor, Structured Development for Real-time Systems, Four Volumes, Prentice Hall, Englewood Cliffs, New Jersey, 1985.

[WATE91] R.C. Waters and Y.M. Tan, "Toward a Design Apprentice: Supporting Reuse and Evolution in Software Design," *ACM SIGSOFT Software Engineering Notes*, Vol. 16, No. 2, April 1991, pp. 33-44.

[WEBS88] D.E. Webster, "Mapping the Design Information Representation Terrain," *Computer*, Vol. 21, No. 12, December 1988, pp. 8-24.

[WIRF90] R. Wirfs-Brock and R. Johnson, "Surveying Current Research in Object-Oriented Design," *Communications of the ACM*, September 1990, pp. 104-124.

[WOOD87] S. Woodfield, et al., "Can Programmers Reuse Software?" *IEEE Software*, July 1987, pp. 52-59.

[YAU86] S.S. Yau and J.P. Tsai, "A Survey of Software Design Techniques," *IEEE Transactions On Software Engineering*, Vol. SE-12 No. 6, June 1986, pp. 713-721.

[YOUR79] E. Yourdon and L.L. Constantine, Structured Design, Prentice-Hall, Englewood Cliffs, NJ, 1979.

APPENDIX A. SOME PROPOSED DESIGN HEURISTICS EXPRESSED AS RULES

This appendix contains a set of design heuristics that might be applied to construct a concurrent design for software from the OCD (object communication diagram) view of a target specification generated from an EDLC (Evolutionary Domain Life Cycle) model. These heuristics can also be applied to any OOA (object-oriented analysis) specification that can be expressed using an OCD view. As with any heuristics that embody human knowledge, the rules specified below do not represent a complete set of design knowledge (indeed, as described in the main text, other design knowledge might include: design-process knowledge, priority-assignment knowledge, task-inversion knowledge, and performance-evaluation knowledge). In addition, further work is needed to verify and validate the proposed rules.

The rules are presented in five categories. Section A.1 contains rules intended to identify active objects from the specification and to encapsulate those objects into various types of tasks. Section A.2 presents rules for assigning passive objects to tasks. Section A.3 defines rules for merging tasks. Section A.4 gives rules for mapping object messages to task interfaces. Section A.5 contains rules for mapping timers, interrupts, and data to task interfaces.

A.1 Active Object Identification Rules

R1: if $object_i$ is a Device Input Object and
the device associated with $object_i$ generates interrupts
then
wrap $object_i$ in an Asynchronous Device Input Task

R2: if $object_i$ is a Device Input Object and
the device associated with $object_i$ is Passive and
the device associated with $object_i$ must be polled
then
wrap $object_i$ in a Periodic Device Input Task

R3: if $object_i$ is a Device Output Object and
the device associated with $object_i$ generates interrupts
then
wrap $object_i$ in an Asynchronous Device Output Task

R4: if $object_i$ is a Device Output Object and
the device associated with $object_i$ is Passive and
the device associated with $object_i$ must be strobed
then
wrap $object_i$ in a Periodic Device Output Task

R5: if $object_i$ is a Control Object
then
wrap $object_i$ in a Control Task

R6: if $object_i$ is a User Role Object
then
wrap $object_i$ in a User Role Task

R14: if $object_i$ is a Device Output Object and
the device associated with $object_i$ is Passive and
the device associated with $object_i$ need not be strobed and
$object_i$ receives messages from more than one other object
then
wrap $object_i$ in a Resource Control Task

R8: if $object_i$ is an Algorithm Object and
$object_i$ must execute periodically and
$object_i$ is not enabled by an $object_j$
then
wrap $object_i$ in a Periodic Internal Task

R9: if $object_i$ is an Algorithm Object and
$object_j$ enables $object_i$ and
$object_i$ has an indeterminate execution time
then
wrap $object_i$ in an Asynchronous Internal Task

R10: if $object_i$ is an Algorithm Object and
$object_j$ enables $object_i$ and
$object_i$ has a period
then
wrap $object_i$ in an Asynchronous-Periodic Internal Task

R11: if $object_i$ is an Algorithm Object and
$object_i$ is not enabled by an $object_j$ and
$object_i$ has an indeterminate execution time
then
wrap $object_i$ in a Background Task

A.2 Passive Object Assignment Rules

R1: if $object_i$ is an Algorithm Object or a Data Abstraction Object and
$object_i$ is not encapsulated in a task and
$object_i$ is accessed from a single $task_j$
then
encapsulate $object_i$ in $task_j$

R2: if $object_i$ is an Algorithm Object or a Data Abstraction Object and
$object_i$ is not encapsulated in a task and
$object_i$ is accessed from multiple tasks
then
denote $object_i$ as a shared object

R3: if $object_i$ is shared and
$object_i$ receives a message from $object_j$ and
$object_j$ is encapsulated in $task_k$
then
denote that $task_k$ references $object_i$

A.3 Active Object Cohesion Rules

R1: if $task_j$ is a Periodic Device Input Task and
$task_k$ is a Periodic Device Input Task and
the period of $task_j$ resonates with the period of $task_k$ and
the importance of $task_j$ is comparable with the importance of $task_k$
then
merge $task_j$ with $task_k$

R2: if $task_j$ is a Periodic Device Output Task and
$task_k$ is a Periodic Device Output Task and
the period of $task_j$ resonates with the period of $task_k$ and
the importance of $task_j$ is comparable with the importance of $task_k$
then
merge $task_j$ with $task_k$

R3: if $task_j$ is a Periodic Internal Task and
$task_k$ is a Periodic Internal Task and
the period of $task_j$ resonates with the period of $task_k$ and
the importance of $task_j$ is comparable with the importance of $task_k$ and
$task_j$ and $task_k$ are related functionally
then
merge $task_j$ with $task_k$

R4: if $task_j$ is a Control Task and
$task_j$ receives messages from $task_k$ and
$task_j$ receives messages from no other task and
$task_k$ is a Periodic Internal Task
then
merge $task_j$ and $task_k$ a single Periodic Internal Task

R5: if $task_j$ is a Control Task and
$task_j$ sends messages to $task_k$ and
$task_k$ is an Asynchronous Internal Task and
$task_k$ must finish before $task_j$ can continue
then
merge $task_j$ and $task_k$ into a single Control Task

R6: if $task_j$ is an Asynchronous-Periodic Internal Task and
$task_k$ is an Asynchronous-Periodic Internal Task and
$task_j$ is enabled by $task_i$ and
$task_k$ is enabled by $task_i$ and
$task_j$ and $task_k$ are mutually exclusive
then
merge $task_j$ and $task_k$

A.4 Rules For Mapping Object Messages To Task Interfaces

To a large extent, the mapping of object messages to task interfaces depends upon the messaging facilities supported by the target, run-time system. Most run-time systems provide message queuing facilities between tasks to handle routine communications and also support a synchronization mechanism that embodies inter-task procedure calls (i.e., tightly-coupled messages, with reply). The existence of other forms of synchronization, such as tightly-coupled messages, without reply, is less certain. When a design requires that some messages exceed others in importance, several techniques might be available to support the requirements. For example, a run-time system might facilitate priority message queuing between tasks. In other run-time systems, multiple queues might be required (coupled to a message servicing discipline) to achieve the same effect. In still other systems, a signalling mechanism might be available to serve such needs.

The rules that follow assume that the run-time system supports: 1) loosely-coupled message queues of a single priority, 2) tightly-coupled message passing, both with and without reply, and 3) an inter-task signalling mechanism where signals are of equal priority. Design heuristics to support other assumptions can be defined. In fact, part of the design process might include selecting the messaging facilities available in the intended, run-time system and then using an appropriate set of design rules for that portion of the design process devoted to defining task interfaces.

R1: if $task_i$ receives $message_m$ from $task_j$ and
the importance of $message_m$ is normal and
$message_m$ requires no synchronization
then
map $message_m$ to a loosely-coupled interface from $task_j$ to $task_i$

R2: if $task_i$ receives $message_m$ from $task_j$ and
the importance of $message_m$ is normal and
$message_m$ requires synchronization and
$message_m$ is not coupled to another message
then
map $message_m$ to a tightly-coupled interface, without reply, from $task_j$ to $task_i$

R3: if $task_i$ receives $message_m$ from $task_j$ and
the importance of $message_m$ is normal and
$message_m$ requires synchronization and
$message_m$ is coupled to another $message_n$
then
map $message_m$ to a tightly-coupled interface, with reply, from $task_j$ to $task_i$ and
map $message_n$ to the reply on the same interface from $task_i$ to $task_j$

R4: if $task_i$ receives $message_m$ from $task_j$ and
$message_m$ has no parameters and
the importance of $message_m$ is high
then
map $message_m$ to an internal event from $task_j$ to $task_i$

A.5 Rules for Mapping Interrupts, Timers, and Data to Task Interfaces

R1: if $task_i$ encapsulates an $object_i$ and
$object_i$ is enabled by a timer and
then
map a system timer event to the event input interface for $task_i$

R2: if $task_i$ encapsulates an $object_i$ and
$object_i$ is enabled by an interrupt from
a $device_i$ associated with $object_i$
then
map the interrupt from $device_i$ to an external event input for $task_i$

R3: if $task_i$ encapsulates an $object_i$ and
$object_i$ is associated with a $device_i$ and
$device_i$ provides system inputs
then
map the input from $device_i$ to a data input for $task_i$

R4: if $task_i$ encapsulates an $object_i$ and
$object_i$ is associated with a $device_i$ and
$device_i$ receives system outputs
then
map the outputs to a data output from $task_i$ to $device_i$